

Notes for Math 3380
Algorithms for Applied Mathematics

Dr. Stephen Graves
The University of Texas at Tyler

Spring 2016

Introduction

When researching the topic for the last chapter of this book, I was struck by the following passage¹ in Yefim Dinitz's discussion of the algorithm which bears his name:

Shortly after the “iron curtain” fell in 1990, an American and a Russian, who had both worked on the development of weapons, met. The American asked, “When you developed the Bomb, how were you able to perform such an enormous amount of computing with your weak computers?”. The Russian responded: “We used better algorithms.”

The message Dinitz expands upon in his paper is that a school of mathematics arose in the USSR favoring strong algorithms designed around carefully-planned data structures to solve computationally intensive problems; in the West, advances in computer speed allowed weaker algorithms to be at least as successful as Soviet algorithms, as the differences were made up in the hardware.

Introductory students should understand this. If you can devise a careful algorithm on paper, you can build a data structure to model it. The process of solving mathematical problems computationally is then two-fold: first, work out an on-paper algorithm which solves the problem; second, determine the method of storing the relevant data for the problem in such a way that implementing the algorithm is efficient and understandable.

This book is divided into four parts. The first is an extremely basic introduction to programming in Python 3.5. There are several reasons Python was chosen rather than another language, but the primary reasons are portability and cost. Python can be run under every operating system (to my knowledge) and is free and open source. The second part of the book applies the material from the first in the development of a robust data structure often used to solve mathematical problems: the matrix. We work from the basic definitions of a matrix through many of the operations of matrix theory, culminating with discussions of the LU and QR decompositions of a matrix. Each of these is used for solving a different class of practical problem. The third unit of the course is a very shallow introduction to cryptography, again with a focus on the data structures used to perform permutation arithmetic. Finally, the text concludes in the fourth part with an introduction to algorithmic graph theory. Various data structures for representing graphs computationally are discussed, and then ignored, as the three presented problems lend themselves very well to particular models rather than general ones.

¹Dinitz, Yefim. “Dinitz’ algorithm: The original version and Even’s version.” *Theoretical Computer Science*. Springer Berlin Heidelberg, 2006. 218-240.

Contents

Contents	iii
I Crash course in programming with Python	1
1 Welcome to Algorithms and Python!	3
1.1 Interactive mode	3
1.2 Python and numbers	4
1.3 Strings	4
2 Lists and iteration	9
2.1 Lists	9
2.2 Modules	11
2.3 for loops	11
2.4 Power tool: list comprehension	13
2.5 tuples	14
3 Conditioning and error handling	17
3.1 Boolean values	17
3.2 Conditional statements	18
3.3 Errors, error handling, and the <code>try</code> statement	19
3.4 <code>break</code> and <code>pass</code>	20
4 Functions and object-oriented programming	23
4.1 Functions in programming	23
4.2 Classes and object-oriented programming	25
4.3 Power tool: operator overloading	29
II Matrix algorithms	33
5 Matrices and their operations	35
5.1 What is a matrix?	35
5.2 How would a computer best represent a matrix?	36
5.3 Beginning matrix operations	37
5.4 Matrix Multiplication	39

6	Vectors and solving systems of equations	43
6.1	Vectors in Euclidean space	43
6.2	Operations on vectors	44
6.3	Motivation for matrix arithmetic	45
6.4	A floating-point problem	48
6.5	Pivoting strategies	49
6.6	Algorithms for Gaussian elimination	50
7	Matrix decomposition: LU decomposition	53
7.1	Elementary matrices	53
7.2	Matrix inverses	55
7.3	LU decomposition	56
7.4	$PA = LU$ factorization	59
8	Matrix decomposition: QR decomposition	61
8.1	Least squares regression	61
8.2	Application of least squares to curves of best fit	64
8.3	Gram-Schmidt orthonormalization	65
8.4	QR factorization	67
8.5	QR factorization by Householder reflectors	68
III	Introduction to cryptography	73
9	Permutations	75
9.1	What is a permutation?	75
9.2	Permutation groups	77
9.3	Cycles and disjoint cycle decomposition	78
9.4	Implementing permutations in Python	80
9.5	A basic permutation class	82
10	Substitution Ciphers	85
10.1	Simple substitution	85
10.2	Polyalphabetic substitution	86
11	Public Key Encryption	93
11.1	Shared secret vs. public key encryption	93
11.2	Kid RSA	94
11.3	Plain RSA	96
IV	Algorithmic graph theory	99
12	Introduction to Graph Theory	101
12.1	Graphs	101
12.2	Representing graphs computationally	104
13	Shortest Paths	107

13.1 Erdős Number	107
13.2 Dijkstra's Algorithm	107
13.3 Implementing Dijkstra's algorithm	110
14 Minimum spanning trees	111
14.1 Weighted graphs & minimum spanning trees	111
14.2 Kruskal's algorithm	111
14.3 Implementing Kruskal's algorithm	114
15 Maximum flow in a capacitated network	117
15.1 Capacitated networks	117
15.2 Max flow	117
15.3 Dinitz' Algorithm	118
15.4 Implementing Dinitz' Algorithm	124

Part I

Crash course in programming with Python

Lesson 1

Welcome to Algorithms and Python!

Goals

1. Gain familiarity with the IDLE environment.
2. Learn some elementary Python syntax and commands.

Special Instructions

If you are using your own laptop, and did not follow the instructions posted on Blackboard, you will have to complete today's lecture on one of the University's computers and then download and install Python 3.5 yourself from <http://python.org>.

Please have IDLE open and running when you're reading notes so that you can actually work examples.

We're going to extensively discuss actual Python code in these notes, so let me determine some conventions. If I write something like `print`, you should understand that it is a Python command. Anything that looks like this:

```
x=13
print(x**2)
```

is a piece of Python code.

1.1 Interactive mode

When you first open IDLE, you should get a report like so:

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

1. WELCOME TO ALGORITHMS AND PYTHON!

From this you can deduce two things: Python is running, and I use a Mac. The `>>>` which appears at the bottom is the *prompt*; it's waiting for you to give Python a command. For today's notes, I'll explicitly tell you when to press Return by showing `↵`.

```
>>> 3 + 7 ↵  
10
```

Congratulations, you've just used Python as a calculator!

1.2 Python and numbers

Python works really well with integers (called `ints`) and floating point numbers (called `floats`), and carefully respects order of operations.

```
>>> 4 * (0 + 4 ** 2 - 1) / 3 ↵  
20.0
```

In the above example, we note two things: first, the `**` operator is our power operator; second, we fed in all `integers` and the returned value was a `float`. This is the behavior of `/`, which is the division operator. Integer division (via the Division Algorithm) uses `//` and `%` for quotient and remainder respectively:

```
>>> 16 // 3 ↵  
5  
>>> 16 % 3 ↵  
1
```

To assign a value to a variable name, we use `=`:

```
>>> spam = 5 ↵  
>>> eggs = spam ↵  
>>> spam = 90.5 ↵  
>>> eggs ↵  
5  
>>> spam ↵  
90.5
```

We see now that first setting `eggs` equal to the value of `spam` and then changing `spam` does not also change `eggs`.

1.3 Strings

Mathematically, a *string* is a finite sequence of symbols from an *alphabet*, which is really just the set of allowable symbols. Python behaves similarly: if you can type something, it's an allowable symbol. Strings in Python are differentiated from variable names and commands by being enclosed in single or double quotes. The only difference between single and double quotes for Python has to do with quotes within strings.

The operation of attaching one string to the end of another string is called *concatenation*, and is carried out in Python using the `+` operator. Integer multiplication of strings produces multiple copies of the string.

```
>>> spam = 'There, he moved!'
>>> eggs = "No he didn't, that was you hitting the cage!"
>>> 3 * spam
'There, he moved!There, he moved!There, he moved!'
>>> spam + eggs
'There, he moved!No he didn't, that was you hitting the cage!'
```

Observe that you must be careful when entering strings: improper use of quotation marks will cause syntax errors.

```
>>> eggs = 'No he didn't, that was you hitting the cage!'
SyntaxError: invalid syntax
```

Another way to avoid this is to *escape* the special character:

```
>>> eggs = 'No he didn\'t, that was you hitting the cage!'
>>> eggs
'No he didn't, that was you hitting the cage!'
```

There are several other special “escape characters,” including `\n` (new line) and `\t` (tab) which will not display when a string is returned, but only when it is **printed**.

```
>>> eggs = 'No he didn\'t, that was you hitting the cage!\n\tI never!'
>>> eggs
'No he didn't, that was you hitting the cage!\n\tI never!'
>>> print(eggs)
No he didn't, that was you hitting the cage!
    I never!
```

The `print(...)` *function* prints a formatted string version of whatever *argument(s)* it is given; if it is given more than one argument, it separates them with a space:

```
>>> print(3, 5, spam)
3 5 There, he moved!
```

The behavior of `print` can be further controlled: if given the parameter `sep='xx'` successive arguments would be separated by `xx` rather than a space; if given the parameter `end=' '` the string is terminated with a single space rather than a new line.

Since strings are actually sequences of printable characters, we should be able to access elements of the sequence individually; in other languages, these pieces are called *characters*,

but Python has no separate character class. They are simply strings of size 1. An extremely important thing to remember about Python is that all sequence-type objects have first index 0. If you wanted to represent the sequence of positive integers less than or equal to 100, $a = (1, 2, 3, 4, \dots, 100)$, the elements of the tuple would be $a_0 = 1$, $a_1 = 2$, $a_2 = 3$, and so on, up to $a_{99} = 100$.

Rather than subscripts, indices in Python are given in square brackets. Trying to index a position outside a string will result in an error.

```
>>> spam = 'this is a string' ↵
>>> spam[0] ↵
't'
>>> spam[5] ↵
'i'
>>> spam[-1] ↵
'g'
>>> spam[42] ↵
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    spam[42]
IndexError: string index out of range
```

Slicing

You can access a substring by *slicing* the string, but you have to be careful how slicing works. The best way to imagine it (as explained in the Python tutorial) is to think of the indices actually marking the dividers between letters in a string:

From Front																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t	h	i	s		i	s		a		s	t	r	i	n	g	
-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	
From Back																

So if this is the string `spam`, then we can see that `spam[0]` gave us the single-letter string starting at index 0, and `spam[-1]` gave us the single-letter string starting at index -1 from the back. If we want to isolate the word `this` we can do it two ways. Since the word `string` appears at the end of `spam`, there are several ways to easily access it:

```
>>> spam[0:4] ↵
'this'
>>> spam[:4] ↵
'this'
>>> spam[10:16] ↵
'string'
>>> spam[10:] ↵
'string'
>>> spam[-6:] ↵
```

```
'string'
```

Negative and positive indices can be mixed and matched, and mixing is handled smartly. Also, Python will accept a slice beyond the length of a string.

```
>>> spam[-11:9] ↵  
'is a'  
>>> spam[8:-11] ↵  
,,  
>>> spam[8:42] ↵  
'a string'
```

You can always find the length of your string using the `len(...)` command:

```
>>> len(spam) ↵  
16
```

Strings are *immutable*

An important thing to note about strings is that you may read individual positions but you may not overwrite them.

```
>>> spam[5]='1' ↵  
Traceback (most recent call last):  
  File "<pyshell#41>", line 1, in <module>  
    spam[5]='1'  
TypeError: 'str' object does not support item assignment
```

In the next lesson we will talk about other *compound* objects which are *mutable*, where you can change elements of the container one at a time.


Lesson 2

Lists and iteration

Goals

1. Define the `list` data type.
2. Discuss the use of modules rather than interactive Python.
3. Define the `for... in...` iteration structure.

Special Instructions

Please have IDLE open and running. In this lesson the  symbol will no longer appear in example code.

2.1 Lists

In the previous lesson, we discussed `string` objects at some length. One unsatisfactory truth of working with strings is that it is impossible to change a single character in the middle of a pre-existing string, without overwriting the value of the string. This behavior occurs because strings are an *immutable* data type.

Our first example of an *mutable compound data type* is the `list`. Lists may contain as elements any other data type, including others which are compound! Syntactically, they are very easy to implement: a list consists of a comma-separated list of objects enclosed in square brackets. The elements can be indexed individually, or sublists can be constructed by slicing, exactly as when using strings.

```
>>> foo = 'this is a string'
>>> bar = [1,2,3,foo,5]
>>> bar
[1, 2, 3, 'this is a string', 5]
>>> bar[-2:]
['this is a string', 5]
```

Also, lists can be added and multiplied by integers:

2. LISTS AND ITERATION

```
>>> [1,2,3] + ['a','b','c']
[1, 2, 3, 'a', 'b', 'c']
>>> 5 * [1,2,3]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Methods

A wonderful feature of Python can be found in its implementation of *methods*¹, predefined functions available within each class of objects. For instance, `bar.reverse()` will reverse the order of the elements of `bar`:

```
>>> bar.reverse()
>>> bar
[5, 'this is a string', 3, 2, 1]
```

Another useful list method is `sort`, which reorders the elements of a list into sorted order, but *only if all the elements can be compared!* For instance, if we sort our list `bar` as it is, we will get an error.

```
>>> bar.sort()
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bar.sort()
TypeError: unorderable types: str() < int()
```

Now, we have noted that lists are mutable, so we can change individually indexed pieces of `bar`. If we replace `bar[1]` with a number, for instance, we can sort `bar` in place using the `bar.sort()` method.

```
>>> bar[1] = 9.4
>>> bar
[5, 9.4, 3, 2, 1]
>>> bar.sort()
>>> bar
[1, 2, 3, 5, 9.4]
```

For more information on what methods are available to a given data type, you can define a variable of that type and then pass that variable name as an argument to the `help` function. This will also give you an introduction to the style of Python's documentation. Try declaring `temp` to be the empty list and then ask Python for `help` about `temp`:

```
>>> temp = [ ]
>>> help(temp)
```

It is always the case that methods available to a variable `my_var` will be accessed via `my_var.method_name(...)`; we'll talk more about this when we get to functional and object-oriented programming in Lesson 4.

¹We'll extensively discuss methods and classes after we've talked about functions, when we briefly discuss object-oriented programming. This will happen just before we start diving into "real math."

2.2 Modules

Everything up until now has focused on using Python as a very powerful calculator. To do anything more sophisticated, we must introduce *modules*. In your IDLE window, select from the menu File → New File. An empty window will open without the familiar `>>>` prompt! As an example of how modules work, let's play with a string method, `swapcase()`.

```
foo = 'Hello World!'
for x in foo:
    print(x.swapcase())
```

Now, save this module as `les2_ex1.py` in a convenient location. Then select “Run Module” from the Run menu. Something interesting will appear in the Python Shell window; for instance, what I see is this:

```
===== RESTART: /Users/sgraves/Documents/Code/python/les2_ex1.py =====
h
E
L
L
O

w
O
R
L
D
!
>>>
```

There are many, many benefits to writing code in modules rather than working extensively in interactive mode. Primary for students is the ability to edit code – anyone who has made a mistake in these first two lessons knows fully well how irritating it is to retype incorrect code in interactive mode. The principle downside of using modules is that any output you wish to see must be **printed**. For instance, you can change the string `foo` to any string you choose, and if you then save and run the module it will swap the upper and lower case characters in this new string.

Since we are still learning basic concepts, we will continue to use interactive mode whenever it is instructive to do so.

2.3 for loops

This example also introduces a new coding structure, called a *for loop*. Looping, or *iteration* as it is properly called, is really the primary task of computers: repeatedly perform tedious tasks. In most programming languages, for loops are structured in such a way that some *index variable* takes values from a starting point to an ending point with a specific step size. Python makes this even more clever by indexing over the elements of a compound data type!

In our example, the data type of `foo` is a string, so its elements are single-letter strings. The loop starts by setting `x` to the value `'H'`. Then it enters the indented *block* of code below the `for` statement. It will run everything in this indented block, and upon reaching the outdent will set `x` to the next value in `foo`.

Here's another module to try: enter this and save it as `les2_ex2.py` in the same directory or folder as your previous example, then run it.

```
bar = "It's just a flesh wound."
n=79 - len(bar)
for i in range(n):
    print(i * ' ' + bar)
```

This introduces the `range` function, which is actually a very special data type. The command `range(10)`, for instance, creates an object which will contain all integers $0 \leq x < 10$ – but not until it is iterated over! You can sort of think of the range function as a promise to provide a list when needed, rather than the list itself. There are actually three ways `range` can be called:

Command	Produces
<code>range(i)</code>	The sequence of integers $0, 1, 2, \dots, i - 1$.
<code>range(i, j)</code>	The sequence of integers $i, i + 1, i + 2, \dots, j - 1$.
<code>range(i, j, k)</code>	The sequence of integers $i, i + k, i + 2k, \dots, i + mk$ where m is the maximum integer such that $i + mk < j$.

Importantly, `range(10)` is precisely the list of valid indices for a `list` containing ten elements.

All loops can be nested within one another, but remember that this makes the number of executions multiply, not just add.

Infinite loops

It is difficult, but not impossible, to get stuck in an “infinite” for loop. To do this, you would have to **append** data to the list over which the loop iterates at least as fast as the iteration progresses. This rapidly uses an incredibly large amount of memory and will bring your computer to a crippling halt.

If you believe that a loop has gotten away from you, you have several remedies. First, try pressing Control-C. This sends a `KeyboardInterrupt` to Python and should stop most runaway processes. If that fails, try restarting the Python Shell. On my various Mac computers, this is achieved by pressing Control-F6. If that fails, you can always quit IDLE. If that fails, you can always force IDLE to quit, using the Windows Task Manager or OS X Activity Monitor. If that fails, you can always restart your computer².

If you'd like to try an example of an infinite loop, enter the following code, and when you get bored of “nothing happening” press Control-C, then evaluate `len(x)`. Here's what

²None of these things will “break your computer,” **all else being equal**. Since Python is a high-level programming language, it is **possible** to “break your computer” using Python code – but you have to be trying to do this rather than it happening accidentally. Even then, in almost every case you only damage the software; a clean installation of your operating system, while time consuming, fixes software problems. This is still not something to worry about. In more than two decades of programming, I've never broken anything to the point where I needed to rebuild my OS.

you'll see, although the speed of your computer and how long you wait will determine the length of `x` when you stop. The purpose of running `x.clear()` at the end is to relieve the “memory pressure” of a large list of integers. I waited until the memory used by Python exceeded 1 GB as reported in the OS X Activity Monitor.

```
>>> x = [0]
>>> for i in x:
    x.append(i)

Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    for i in x:
KeyboardInterrupt
>>> len(x)
144887010
>>> x.clear()
```

2.4 Power tool: list comprehension

Suppose you had a list `foo` of numbers, and you wanted to set `bar` equal to the list of their squares. We could easily perform this task with a for loop and the `append` list method, which adds the argument to the end of the list.

```
>>> foo = [1, 3, 5, 7, 9]
>>> bar = []
>>> for x in foo:
    bar.append(x**2)

>>> bar
[1, 9, 25, 49, 81]
```

However, Python provides a much more elegant way to do the same thing, called *list comprehension*. Essentially this provides a way to create a list by successively performing the same operation (in this case squaring) on all the elements of another list.

```
>>> foo = [1, 3, 5, 7, 9]
>>> bar = [ x ** 2 for x in foo ]
>>> bar
[1, 9, 25, 49, 81]
```

This seems like a trivial example, but what about this: we want to approximate a parabola $y = x^2$ by using 101 points from 0 to 1. How do we calculate those points?

```
>>> pts = [(x, x ** 2) for x in [i / 100 for i in range(101)]]
```

To be more sophisticated, what if we want to square $x_0 = a$ and $x_n = b$, and all the n equally-spaced values between?

2. LISTS AND ITERATION

```
>>> a, b, n = 5, 8, 10
>>> pts = [(x, x ** 2) for x in [a + (b - a) * i / n for i in range(n + 1)]]
>>> pts
[(5.0, 25.0), (5.3, 28.09), (5.6, 31.359999999999996), (5.9, 34.81), (6.2, 38.44
000000000000005), (6.5, 42.25), (6.8, 46.239999999999995), (7.1, 50.41), (7.4, 54.
76000000000000005), (7.7, 59.290000000000006), (8.0, 64.0)]
```

There are two additional ideas demonstrated in this example code. The first occurs in the very first line of the code sample.

```
>>> a, b, n = 5, 8, 10
```

This line is a *multiple assignment*, where the *tuple*³ of values (5, 8, 10) is assigned to the tuple of variables (*a, b, n*). Second, arithmetic with *floating point* numbers is inexact arithmetic: we can see analytically that $6.2^2 = 38.44$ but

```
>>> (6.2)**2 - 38.44 == 0
False
```

While this is not a course on error analysis and correcting for floating point arithmetic, we will try to avoid this type of error when possible.

2.5 tuples

A **tuple** is another compound data type which looks almost exactly like a **list**, but is immutable. Elements of tuples can be any data type, but once a **tuple** is assigned its elements cannot be changed without overwriting the whole **tuple**.

```
>>> spam = (1, 2, 3, 4)
>>> type(spam)
<class 'tuple'>
>>> spam[3]=2
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    spam[3]=2
TypeError: 'tuple' object does not support item assignment
>>> eggs=list(spam)
>>> print(eggs, type(eggs))
[1, 2, 3, 4] <class 'list'>
>>> spam == tuple(eggs)
True
```

As you can see, you can directly convert between items of type **list** and **tuple**. Unfortunately, **tuple** comprehension doesn't work like **list** comprehension.

```
>>> mylist = [ x**2 for x in range(10)]
>>> mytuple = ( x**2 for x in range(10) )
>>> mylist
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

³We will later learn about **tuple**, which is actually a Python data type related to lists. Here we use the mathematical definition of tuple: a finite ordered sequence.

```
>>> type(mylist)
<class 'list'>
>>> mytuple
<generator object <genexpr> at 0x1046568e0>
>>> type(mytuple)
<class 'generator'>
>>> tuple(mytuple)
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

The **generator** object created by this attempted **tuple** comprehension is very similar to the **range** objects. We won't stress out too much about the properties of **range** and **generator** objects, and will try to avoid them where possible.

Lesson 3

Conditioning and error handling

Goals

1. Define the `bool` data type.
2. Discuss conditional statements.
3. Discuss error handling via `try:... except:... blocks`.

3.1 Boolean values

In the language of mathematics, a *statement* is any expression which may be either true or false. These two values, `True` and `False` are the fundamental values of data type `bool`, and are likewise the fundamental values of *Boolean algebra*¹. Boolean algebra is an algebraic system formalizing the same concepts as propositional calculus, which is in turn the formal language of logic and proof taught in Foundations (Math 3425). Here we use a very simple subset of Boolean algebra: expressions in Python which evaluate to `True` or `False` are the basis of all *branching processes*, by which computers are instructed on how to make decisions.

There are many, many Python commands which return `bool` values. The comparison operators are first among them.

Operator	Meaning
<code>x == y</code>	Is x formally equivalent to y ?
<code>x != y</code>	is x formally different from y ?
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal to
<code>x > y</code>	Greater than
<code>x >= y</code>	greater than or equal to

For the ordered comparison operators to work, the objects to be compared must have an order. It may be surprising what data types are ordered in Python and thus can be compared using `<` and `>`.

Additionally, Boolean statements may be combined and modified using the logical operations. Assume in the following table that `x` and `y` are of type `bool`.

¹Name after George Boole, author of *The Laws of Thought* (1854).

Name	Usage
Negation	<code>not x</code>
Conjunction	<code>x and y</code>
Disjunction	<code>x or y</code>

These may be combined using parentheses to follow all the rules of logic.

3.2 Conditional statements

while loops

The first type of conditional statement we will consider is the “other” kind of iteration: the **while** loop. It is much, much easier to enter an infinite while loop, but they generally create less memory pressure than infinite for loops. Here’s the easiest do-nothing infinite while loop:

```
>>> while True:
    None
```

The structure of a **while** loop is uncomplicated, and its meaning follows immediately from its structure: while some statement evaluates to **True**, execute an indented block of code. Upon reaching the end of the indented block, check if the statement is still true; if so, repeat the block, and if not, move on beyond the loop. We can use a while loop to compose an annoying SMS by counting until we run out of room. Since text messages (prior to smart phones) were limited to 160 characters, we save the following as `les3_ex1.py` and execute it.

```
s = ''
i = 1
limit = 160
while len(str(i))+len(s)+1 < limit:
    s = s + str(i) + ' '
    i = i + 1
print(s)
```

A common error when using while loops in this way is failure to *increment your counter*. This is the `i = i + 1` line. What would happen if we left it off? Would this loop run forever? There is a shorthand for incrementing a variable: for every data type where addition makes sense, you can use `oldvar += newvar` instead of `oldvar = oldvar + newvar` as long as the data types of `oldvar` and `newvar` match. This means I could have written the above example as follows.

```
s = ''
i = 1
limit = 160
while len(str(i))+len(s)+1 < limit:
    s += str(i) + ' '
    i += 1
print(s)
```


if statements

Often we only wish to execute a block of code if some condition is met. The natural language way that this would be expressed is something like “If the condition is met, then do task A. Otherwise, do task B.” This is available in Python using `if:... else:...` blocks. Here’s an example which you should save as `les3_ex2.py`.

```
s = input('Please type something here and then press return: ')
if len(s)%2==0:
    t='even'
else:
    t='odd'
print('You typed an '+t+' number of characters.')
```

Just like loops, `if...:` statements are block structures; the block is complete when the level of indentation returns to the same level as the original `if` statement.

It is frequently the case that `if` blocks need to be nested, and just as often the case where if the test fails, we want to test something else. This is called an *else if*, implemented in Python by the `elif` command. We’ll use the following code, `les3_ex3.py`, to demonstrate this and to introduce another type of conditional statement.

```
while True:
    try:
        s = input('Please type an integer: ')
        n = int(s)
        if n%3==1:
            t='one greater than '
        elif n%3==2:
            t='one less than '
        else:
            t=''
        print('Your number is '+t+'a multiple of 3.')
        break
    except ValueError as err:
        print('!!! That is not an integer!!!',err)
    except:
        raise
```

3.3 Errors, error handling, and the `try` statement

There are several ways that executing a program can go wrong. The best type of error is a *syntax error*, usually caused by a typo in your code. These won’t even run, and IDLE is very good at detecting them. The second best type of error is an *exception*. Exceptions are errors which are **raised** by doing something which is syntactically correct in code but violates something about the way Python works. Here’s a simple demonstration – you can’t add a string and an integer.

```
>>> s='123'
>>> t=4
>>> s+t
```

3. CONDITIONING AND ERROR HANDLING

```
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    s+t
TypeError: Can't convert 'int' object to str implicitly
>>> t+s
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    t+s
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

How these errors occur is actually built in to the definitions of type `int` and type `str`. Somewhere in the `builtin` code for these types, a command reading `raise(TypeError('...'))` occurs. The *exception* is a *type error*, where you are trying to combine data types which are incompatible. There are lots of predefined exceptions: `TypeError`, `ValueError`, and `NameError` are the three you will most commonly see, followed by `ZeroDivisionError` when you aren't careful in your programming.

In `les3_ex3.py`, I used the `ValueError` which is raised by trying to force the conversion of a non-integer string to an integer. Python is able to correctly handle the conversion of a string like `'12345'` to an integer, but to ask “what is the integer form of `'abc'`?” is a meaningless question. The `try: ... except: ...` block is a very specialized type of `if...then` structure. Python will run the code inside a `try:` block, and if an error is raised will jump to the first `except...:` statement. If the type of the exception matches, that block will execute. The final block of `les3_ex3.py` is

```
except:
    raise
```

which terminates the `try:... except:... structure` by *re-raising any unhandled exception*. This allows you to press Ctrl-C to exit the `while True:` loop, for instance. This is also the behavior of the `break` command: it makes the program immediately exit the innermost enclosing loop.

The raising and handling of exceptions is often treated as an advanced programming topic, but is a useful tool even at the introductory level to help understand how programming errors are made, and how to avoid or mitigate them.

3.4 break and pass

Two special commands to mention explicitly are `break` and `pass`. The first of these, `break`, causes the Python interpreter to immediately exit the current loop. If `break` is called outside a loop, a `SyntaxError` arises:

```
>>> break
SyntaxError: 'break' outside loop
```

We often use this to prematurely exit `for` or `while` loops. Sometimes the test condition for a `while` loop is not easy to explain, and `break` can be used to exit the loop at the appropriate time. In these cases, you can usually come up with some sort of way to avoid using `break` if you desire.

```
>>> while True:
    break

>>>
```

On the other hand, sometimes while writing code you need some “placeholder” code – for instance, you’re trying to write a complicated if-else block, and you want to work on the `else:` part first. The `pass` command is a do-nothing command. In fact, it does so little that you can’t even get `help` on it:

```
>>> help(pass)
SyntaxError: invalid syntax
```

Again, there are other ways to effect the same thing (often just `printing` a debug message, for instance), but `pass` is still occasionally useful.

Lesson 4

Functions and object-oriented programming

Goals

1. Discuss functions defined both by `def ...:` and as `lambda` functions.
2. Introduce classes.
3. Discuss operator overloading.

4.1 Functions in programming

A mathematical function is a well-defined relation between two sets, if you've taken Foundations or other proof-based courses. A function in computer programming is as well, although it may not often seem like it.

In programming, functions are necessary when you need to repeatedly perform the same algorithm, perhaps on a variety of inputs. Input values to a function are called its *arguments*, and we will use the terminology of *passing arguments to a function*. Every Python function also has an output value, although sometimes its output is `None`.

lambda expressions

The simplest type of function in Python is called a `lambda` expression. Let's say we want to define a function f which takes two arguments, n and x , and returns nx . Mathematically, this is simple: let $f(n, x) = nx$. When we need to define such a simple function for our own use in a Python program, we can define f in almost the same way.

```
>>> f = lambda n,x: n*x
>>> f(4,12)
48
>>> f(4,'12')
'12121212'
>>> f([4],'12')
Traceback (most recent call last):
  File "<pyshell#92>", line 1, in <module>
```

4. FUNCTIONS AND OBJECT-ORIENTED PROGRAMMING

```
f([4], '12')
File "<pyshell#89>", line 1, in <lambda>
  f = lambda n,x: n*x
TypeError: can't multiply sequence by non-int of type 'str'
```

There is a limitation: `lambda` expressions are not block structures, and if there is more than one expression following the `lambda ... :` a syntax error will occur. In fact, trying to enter an indented block by pressing enter after the colon raises a syntax error!

```
>>> g = lambda x:
SyntaxError: invalid syntax
```

The variable names which occur after the keyword `lambda` and before the colon are the tuple of arguments to the function; the expression after the colon is what is evaluated based upon those arguments, and whatever is evaluated is returned as the function's output.

defining functions

The long-form way to define a function, and the way that methods of classes will be defined later in the lesson, is using the `def` command. The syntax is no more complicated than anything else we've worked with so far. Save the following as `les4_ex1.py` and run it in IDLE.

```
def fib(n):
    '''A mysterious function appears!

    The argument n must be a positive integer.
    '''
    if (type(n)!=type(1)) or (type(n)==type(1) and n<0):
        raise TypeError('function fib expects positive integer')
    seq = [1,1]
    while len(seq)<n:
        seq.append(seq[-1]+seq[-2])
    return seq[-1]
```

You'll notice that when you run this through IDLE, nothing appears to happen! However, something important has occurred: try typing `help(fib)` at the prompt.

```
>>> help(fib)
Help on function fib in module __main__:

fib(n)
  A mysterious function appears!

  The argument n must be a positive integer.
```

So we see that now there is a function called `fib`, and if we pass a positive integer to it, something should happen.

Let's look at the code you've saved as `les4_ex1.py` once again. The first expression is `def fib(n):`, which tells Python to define a function named `fib` in the current context and let it have one argument, `n`, which can be used inside the function. Moving into the `def` block, the first expression we encounter is a multiline string! This is very good programming practice in Python, and it's called the *documentation string*, or *docstring* for short. It is good programming style¹ to include a docstring, since it enables the `help` function to tell something about the defined function. I won't go into the formatting of the docstring here: see the Python Tutorial.

Moving into the actual code of our function, we see that the very first thing it does is checks the type and the value of the argument, and if the type is wrong or the integer is negative, a `TypeError` is raised. You can think of this as "bulletproofing" our function: if you're writing code for other people to use, you must be very certain that they cannot through malice or ignorance pass an argument to your function which will fundamentally break stuff². Type and value checking the input to a function is a good way to do this. If you are writing code just for yourself and know that nothing seriously bad will happen if you accidentally pass a float or a string instead of an integer, you will often skip this step.

The next line instantiates a list, `seq`, and the following `while` loop makes use of the `append` method of the `list` class to lengthen the list to the length passed as argument `n`. Once `len(seq) >= n`, the loop exits and the last entry of `seq` is `returned`.

Generally speaking, this structure is what every function will follow! You'll pass in some arguments, maybe do some type checking, and then run an algorithm on the arguments. When you complete the algorithm, you return something. One special note is that even functions which contain no `return` statement return a value – it is the special value `None` of data type `NoneType`. If you try to execute the expression `None`, nothing happens.

```
>>> None
>>>
```

You've already used a function which returns `None`: the `print` function.

```
>>> print(print(5))
5
None
>>>
```

4.2 Classes and object-oriented programming

I'm not going to go into a deep philosophical discussion about the different "programming paradigms," of which *object-oriented programming* is one. We'll treat this in a very intro-

¹Yes, there are such things as good programming style and bad programming style. You can consider it two ways. The first way is that writing code in a nice, consistent style makes it easier for you to later relearn what you've done. The second is that I will have to read code that you write, and if it is horribly difficult to decipher because you haven't been nice to me, I will have a harder time being nice to you! The "manual" of good style for Python is generally accepted to be what is documented in Python Enhancement Proposal 8 (PEP 8). This can be found at <https://www.python.org/dev/peps/pep-0008/>. A very brief overview of that style guide is found at <https://docs.python.org/3/tutorial/controlflow.html#intermezzo-coding-style>.

²This is a technical term.

ductory manner: complicated data types are best considered as a *class* of variables. All the *objects* of the same class should have the same operations which can be performed upon them; these are *methods* of the class. The parameters of objects in the class we will call *attributes*, in keeping with Python terminology.

So we could define a class of complex numbers of the form $a + ib$ where $a, b \in \mathbb{R}$. Arithmetic of complex numbers is different from real numbers, because we have to remember to treat the special number $i = \sqrt{-1}$ accordingly. Mathematically speaking, this means that a complex number $a + ib$ is comprised of an ordered pair of real numbers (a, b) , and we must define carefully what are the algorithms for addition, multiplication, etc.

In Python, it makes sense to define a class of complex numbers. Let's start a new file, `les4_ex2.py`. Since we want to be very careful about what a complex number is, we'll start simply.

```
class ComplexNumber:
    """A complex number class.
    """
```

If this is all we put in our file, nothing interesting happens except that we can now create a variable of type `ComplexNumber` like so:

```
>>> x = ComplexNumber()
>>> type(x)
<class '__main__.ComplexNumber'>
>>> x
<__main__.ComplexNumber object at 0x10367bbe0>
```

This isn't very interesting: when we try to use our `ComplexNumber`, all that Python knows is its type and where in the system's memory it lives. This is because we haven't defined any way for a `ComplexNumber` to be *initialized*. There are two methods that you should always define for a new data class, and initialization is the first of these. In order to know that $x = a + ib$ is a particular complex number, we must know specifically two attributes: its real part a and its imaginary part b . Here's how we will do this:

```
class ComplexNumber:
    """A complex number class.
    """
    def __init__(self, a, b):
        if not (type(a) in [type(0), type(0.1)] and
                type(b) in [type(0), type(0.1)]):
            raise TypeError("ComplexNumber(a,b) expects a and b to both be of \"\
                            + " type 'int' or type 'float'."")
        else:
            self.real_part = a
            self.imag_part = b
```

Here we have a very strange name for a function: `__init__` with three arguments, `self`, `a`, and `b`. The first of these, `self`, is special and should be the first argument of every method of a class. This makes much more sense later when we want to define other methods. We also see that, after the error-checking is done, we assign attributes `real_part`

and `imag_part` to `self` by using `self.real_part` and `self.imag_part` as variables. After `__init__` executes, every attribute of the object being initialized must have a value, even if that value is `None`.

Now we need to get to the other mandatory³ method. This is the *representation* of the object, implemented in Python as the `__repr__` method⁴. This method must return an unambiguous string representation of the complex number.

For complex numbers, this is incredibly easy. Complex numbers are in bijective correspondence⁵ with ordered pairs of real numbers, so the *canonical* representation of a complex number $x = a + iy$ is as the ordered pair (a, b) . Let's extend `les4_ex2.py` to the following:

```
class ComplexNumber:
    ...

    def __repr__(self):
        return str( (self.real_part, self.imag_part) )
```

Now if we save and rerun our module, we can actually assign values to a variable of type `ComplexNumber` and also see what is the value of the variable!

```
>>> x = ComplexNumber(1,3.4)
>>> x
(1, 3.4)
>>> type(x)
<class '__main__.ComplexNumber'>
```

Optionally, you can define the `__str__` method for the class, which defines a “human-readable” format for objects in the class. We might do so as follows.

```
class ComplexNumber:
    ...

    def __str__(self):
        (a, b) = (self.real_part, self.imag_part)
        if a != 0:
            s = str(a)
            if b < 0:
                s += ' - I*(' + str(-b) + ')'
            elif b > 0:
                s += ' + I*(' + str(b) + ')'
        else:
            if b < 0:
                s = '-I*(' + str(b) + ')'
            else:
                s = 'I*(' + str(b) + ')'
        return s
```

Now we can call this function

³Not Python-mandatory, because Python doesn't require this. Instead, it's just extremely bad programming etiquette.

⁴I'm going to pronounce this as “repper,” which rhymes with “pepper.”

⁵Take Foundations!

Adding more methods

It's very easy to add new methods to a class, since they're just functions defined inside the block structure of the class. Let's say we want to be able to determine the magnitude (called a *norm* in mathematics) of a complex number `x` by calling the method `x.norm()`. We'll need to add one line to the beginning of our module, and then we can add our function `norm`. Here's what your file `les4_ex2.py` should now look like:

```
from math import sqrt

class ComplexNumber:
    """A complex number class.
    """
    def __init__(self,a,b):
        if not (type(a) in [type(0), type(0.1)] and
                type(b) in [type(0), type(0.1)]):
            raise TypeError("ComplexNumber(a,b) expects a and b to both be of\"
                            + " type 'int' or type 'float'.")
        else:
            self.real_part = a
            self.imag_part = b

    def __repr__(self):
        return str( (self.real_part, self.imag_part) )

    def __str__(self):
        (a, b) = (self.real_part, self.imag_part)
        if a != 0:
            s = str(a)
            if b<0:
                s += ' - I*(' + str(-b) + ')'
            elif b>0:
                s += ' + I*(' + str(b) + ')'
        else:
            if b<0:
                s = '-I*(' + str(b) + ')'
            else:
                s = 'I*(' + str(b) + ')'
        return s

    def norm(self):
        return sqrt(self.real_part**2 + self.imag_part**2)
```

The line at the beginning, `from math import sqrt`, tells Python to find a module named `math` in either the current directory or Python's module search, and allow us to access the function `sqrt`. Note that if we had already defined a function or variable as `sqrt`, this import command would have overwritten our function or variable! To work around this, we could instead have had our first line be simply `import math`, but then when we wanted to use the function `sqrt` from the `math` module, we would have needed to type `return math.sqrt(self.real_part**2 + self.imag_part**2)`.

What else should we define for complex numbers? It would be nice to be able to access the real and imaginary parts of a complex number, so we'll define `re` and `im` methods.

```

from math import sqrt

class ComplexNumber:
    ...
    def re(self):
        return self.real_part

    def im(self):
        return self.imag_part

```

We should include our arithmetic operations as well. To keep things simple, we'll only define addition and multiplication here, and leave the others as an exercise. Python understands that $3 + 5$ and $5 + 3$ should both be 8, but maybe we want to be lazy with a variables x and y of type `ComplexNumber` and have all of $x + y$, $3 + x$, $x + 3$, $3.1 + x$, and $x + 3.1$ execute correctly. Interestingly, $3 + x$ and $3.1 + x$ will be handled identically, but $x + y$, $3 + x$, and $x + 3$ are all distinct!

4.3 Power tool: operator overloading

Operator overloading is easy for a human and hard for a computer. A person can immediately see the symbol $+$ and know from the context how to interpret it. For instance, we can immediately understand that there is a difference between “ $3 + 5$ ” and “Alice + Bob.” A computer has to be told how to handle each of them separately! This is handled by *overloading* the operator of $+$. Python does this behind the curtain. Every data type for which $+$ makes sense must have in its class definition a method called `__add__`. In fact, usually it is implemented as `def __add__(self, other):`, and then the return value is `self + other`. It is generally also smart to define a `__radd__` method to return `other + self`, and for numerical quantities they should be equal⁶.

Complex addition

What is the algorithm for complex addition? That is, if I defined $x = a + ib$ and $y = c + id$ for some $a, b, c, d \in \mathbb{R}$, what does it mean to write $x + y$? If we treat $x = a + ib$ and $y = c + id$ as polynomials in variable i with real coefficients, we can use the rules for polynomials and obtain

$$\begin{aligned}
 x + y &= (a + ib) + (c + id) \\
 &= a + c + ib + id \\
 &= (a + c) + i(b + d).
 \end{aligned}$$

Since it makes sense for complex numbers and real numbers to follow the same rules, this is a sensible definition for the algorithm of addition.

Algorithm 4.1. (Complex Addition) Let x be a `ComplexNumber` with representation (a, b) and let y be some other variable.

⁶Addition is almost universally a commutative operation.

1. If `y` is a `ComplexNumber` with representation `(c,d)`, return `ComplexNumber(a + c, b + d)`.
2. If not, but `y` is either an `int` or a `float`, return `ComplexNumber(a + y, b)`.
3. Otherwise, raise a `TypeError`.

For complex numbers, we update `les4_ex2.py` as follows to implement addition and right addition.

```
from math import sqrt

class ComplexNumber:
    ...
    def __add__(self, other):
        if type(other) == type(self):
            return ComplexNumber(self.real_part + other.re(),
                                self.imag_part + other.im())
        elif type(other) in [type(1), type(0.1)]:
            return ComplexNumber(other + self.real_part, self.imag_part)

    def __radd__(self, other):
        return self.__add__(other)
```

Complex Multiplication

Complex multiplication, like complex addition, follows naturally from the behavior of polynomials with real coefficients. Again letting $x = a + ib$ and $y = c + id$ be complex numbers, we see that

$$\begin{aligned} xy &= (a + ib)(c + id) \\ &= ac + ibc + iad + i^2bd \\ &= (ac - bd) + i(bc + ad). \end{aligned}$$

If it so happens that $d = 0$, then $xy = ac + ibc$. Again, we can use this to determine the algorithm for complex multiplication.

Algorithm 4.2. (Complex Multiplication) Let `x` be a `ComplexNumber` with representation `(a,b)` and let `y` be some other variable.

1. If `y` is a `ComplexNumber`, return `ComplexNumber(a * c - b * d, a * d + b * c)`
2. If not, but `y` is either an `int` or a `float`, return `ComplexNumber(a*y, b*y)`
3. Otherwise, raise a `TypeError`.

To implement this is similar to the implementation of addition: there are builtin operators for both left and right multiplication (which is important since multiplication is often not commutative) which can be overloaded, named `__mul__` and `__rmul__`. Including them, here is the final, full version of `les4_ex2.py`.

```
from math import sqrt

class ComplexNumber:
    """A complex number class.
    """
    def __init__(self,a,b):
        if not (type(a) in [type(0), type(0.1)] and
                type(b) in [type(0), type(0.1)]):
            raise TypeError("ComplexNumber(a,b) expects a and b to both be of\"
                            + " type 'int' or type 'float'.")
        else:
            self.real_part = a
            self.imag_part = b

    def __repr__(self):
        return str( (self.real_part, self.imag_part) )

    def __str__(self):
        (a, b) = (self.real_part, self.imag_part)
        if a != 0:
            s = str(a)
            if b<0:
                s += ' - I*(' + str(-b) + ')'
            elif b>0:
                s += ' + I*(' + str(b) + ')'
        else:
            if b<0:
                s = '-I*(' + str(b) + ')'
            else:
                s = 'I*(' + str(b) + ')'
        return s

    def norm(self):
        return sqrt(self.real_part**2 + self.imag_part**2)

    def re(self):
        return self.real_part

    def im(self):
        return self.imag_part

    def __add__(self,other):
        (a, b) = (self.real_part, self.img_part)
        if type(other)==type(self):
            (c, d) = (other.re(), other.im())
            return ComplexNumber(a + c, b + d)
        elif type(other) in [type(1), type(0.1)]:
            return ComplexNumber(other + a, b)
        else:
            raise TypeError("No method is defined for addition of \"
                            + "ComplexNumber + '\" \
                            + other.__class__.__name__() + '\"")
```

```
def __radd__(self, other):
    return self.__add__(other)

def __mul__(self, other):
    (a, b) = (self.real_part, self.imag_part)
    if type(other)==type(self):
        (c, d) = (other.re(), other.im())
        return ComplexNumber(a * c - b * d, a * d + b * c)
    elif type(other) in [type(0), type(0.1)]:
        return ComplexNumber(a * other, b * other)
    else:
        raise TypeError("No method is defined for addition of "\
            + "ComplexNumber + '" + \
            + other.__class__.__name__() + "'")

def __rmul__(self, other):
    return self.__mul__(other)
```

Overloading other operators

There are many, many operators and methods which you will want to overload to make robust classes. In fact, almost everything can be overloaded if you can determine how Python handles an operator behind the scenes. For numerical data types, there's an easy way to see what the “normal” operators are: type `help(float)` and read the documentation for the `float` class! In fact, you can often inherit methods from “parent” classes if the operations are already correct.

“It’s a trap!”

In fact, we didn’t need to do any of this work to implement complex numbers. Try `help(complex)`. Following a convention from engineering and the sciences, $j = \sqrt{-1}$, and entering `z = 5-3j` produces a complex number with full functionality. We won’t be saving our `ComplexNumber` class for further use.

Part II

Matrix algorithms

Lesson 5

Matrices and their operations

Goals

Special Instructions

5.1 What is a matrix?

A *matrix* is a rectangular grid of elements¹; in our investigation, we will limit the elements to real numbers. For instance, and by way of demonstrating the form we will use to write a matrix,

$$A = \begin{bmatrix} \pi & e & 6.2 \\ \frac{1}{3} & 7 & -42 \end{bmatrix}$$

and

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

are matrices. Neither

$$C = \begin{bmatrix} 5 & 8 \\ 2 & 3 & 0 \end{bmatrix}$$

nor

$$D = \begin{bmatrix} \text{Eric} & \text{John} \\ \text{Graham} & \text{Terry} \end{bmatrix}$$

are matrices, the first because it is non-rectangular and the second because the names of members of Monty Python are not numbers.

Matrices have many, many interesting properties. These can be investigated at length in a course on Matrix Theory (like Math 3203: Matrix Methods) or better yet in a course on Linear Algebra (like Math 3315: Linear Algebra). We will take an approach much more like Matrix Theory than Linear Algebra so that we can avoid most or all of the “proofiness.”

¹There’s a much more technical definition, but for our purposes this is sufficient.

5.2 How would a computer best represent a matrix?

There is a sort of natural way to represent a matrix algorithmically, which arises from the way we refer to elements of a matrix. For instance, if we are working with a matrix A and are interested in the element of A in the i^{th} row and j^{th} column², we might logically refer to it mathematically as the element $a_{i,j}$. The number of rows of A is its *row dimension*, and the number of columns of A is its *column dimension*. Hence if we have

$$A = \begin{bmatrix} \pi & e & 6.2 \\ \frac{1}{3} & 7 & -42 \end{bmatrix}$$

we would say that A has row dimension 2 and column dimension 3, or more concisely say that A is a 2×3 matrix. Row dimension is always listed first in this notation.

Since we index the element in the i^{th} row and j^{th} column as $a_{i,j}$, it seems natural that the i^{th} row should be something like a list. In fact, since we won't want to overwrite individual elements of a matrix, we'll use a tuple of tuples to store the elements. Without any validation of input, this would be easy enough. However, if we want to be careful, there are many things which could go wrong in trying to input a matrix which we want to protect against – after all, we might use this matrix class later for something more fun!

Here's a brief list of what could be incorrect input for a matrix:

1. The value input could not be have any elements (not a compound data type).
2. The elements of the input value could not be compound types.
3. The first element of the input could be an empty container, like `[]`.
4. The length of the elements of the inputs could be different.
5. The innermost elements could be non-numeric.

All of these errors are addressed in the following code, which you should save as `Matrix.py`.

```
class Matrix:
    """A matrix class.
    """
    def __init__(self, grid):
        input_err = '\n\nMatrix class requires rectangular grid of numbers '+\
            'as input.\n\n'
        try:
            # will raise error if not compound
            self._coldim = len(grid[0])
            if self._coldim==0: # empty first row
                raise ValueError(input_err)
            self._data = []
            for row in grid:
                # check row length and that all row elements are numbers
                if (len(row)!=self._coldim) or \
                    not all([type(x) in [float,int] for x in row]):
                    raise ValueError(input_err[2:-2])
                else:
                    self._data.append(tuple(row))
        except:
            print(input_err)
```

²By convention rows are horizontal and columns vertical.

```

        raise
        self._data = tuple(self._data)
        self._rowdim = len(grid)

    def __repr__(self):
        return str(self._data)

```

This code, like many, introduces a new command, `all`. This is a special boolean function which takes a compound object as its only argument. The effect of running `all` on a list `spam` would be to return `True` if and only if `bool(x)` returns `True` for every `x` in `spam`. We won't worry about producing a prettier output than provided by `Matrix.__repr__` at this point, so we'll leave `Matrix.__str__` undefined.

5.3 Beginning matrix operations

The first matrix operations we will define are more about getting information from a matrix that we have represented than about combining matrices or changing their contents. First, we discussed above the notation of $m \times n$ to denote that a matrix has m rows and n columns. It will be necessary for many of our other methods that we be able to determine the *dimensions* of a matrix, and we don't want to access the “hidden” variables directly.

Exercise 5.1. Write a method `dims` in the `Matrix` class which returns the dimensions of the matrix as a tuple, and add the function to `Matrix.py`.

A slight problem between the notation for matrices in mathematics and the implementation of matrices in Python has to do with indexing of the dimensions. Specifically, the $a_{i,j}$ element of a matrix A would be stored as `A._data[i-1][j-1]`. The builtin method which allows us to use the “bracket notation” for indexing of lists, strings, etc., is `__getitem__`. Likewise, those compound data types which are *mutable* have another method defined, `__setitem__`. We want to define only the first of these, and we will use Python indexing – in fact, the element $a_{i,j}$ will be indexed as `A[i-1,j-1]`. Add the following to `Matrix.py`.

```

def __getitem__(self, coords):
    if type(coords)!=type( (1,1) ):
        raise TypeError('Matrices require a tuple for indexing.')
    i,j = coords
    return self._data[i][j]

```

Now that we can both determine the dimensions of a matrix and read off its elements directly, we are ready to start building new matrices from old. The first way we will do this is via *scalar multiplication*. A *scalar* is a member of the set of numbers with which the matrix is filled – for our purposes, these are *real numbers*, implemented as `ints` and `floats`. As a matter of notation, we will refer to a matrix A whose element in the i^{th} row and j^{th} column is $a_{i,j}$ by writing $A = [a_{i,j}]$.

Definition 5.2. Let k be a real number and $A = [a_{i,j}]$ a real matrix. Then the *scalar product of A by k* is the real matrix $kA = [k \cdot a_{i,j}]$. That is, the elements of kA are the elements of A , each multiplied by k .

Example 5.3. As an example, consider

$$A = \begin{bmatrix} 3 & 4 & 6 \\ 5 & 9 & 12 \end{bmatrix} \text{ and } k = \frac{1}{2}.$$

Then

$$A \cdot k = \begin{bmatrix} \frac{3}{2} & 2 & 3 \\ \frac{5}{2} & \frac{9}{2} & 6 \end{bmatrix} = k \cdot A.$$

<

Unfortunately, we're going to want to use the same operator (`*`, defined in `__mul__` and `__rmul__`) for scalar multiplication, which we've just defined, and *matrix multiplication*, which is much more complicated. We'll handle this inside the `__mul__` method by an `if` statement, and we'll let the case where both `Self` and `other` have type `Matrix` just `pass` for now.

Exercise 5.4. Here's the code for scalar multiplication, with an important piece missing. Notice the `pass` statement inside the first `if` block – this will be replaced later when we define an algorithm for matrix multiplication.

```
def __mul__(self, other):
    mult_err = 'Matrices may only be multiplied by scalars, or '+'\
               'by matrices of appropriate dimensions.'
    if type(self) == type(other):
        pass
    elif type(other) in [type(1), type(0.1)]:
        new_grid = "" "AN IMPORTANT PIECE IS MISSING RIGHT HERE!!"" # <=====
        return Matrix(new_grid)
    else:
        raise TypeError(mult_err)
```

Correct the missing piece with a nested list comprehension. Put your corrected version into `Matrix.py`.

Now we'll move to the first way to combine two matrices: matrix addition. A critical piece which cannot be overlooked is that only matrices of the same dimensions can be added – addition of matrices with different dimensions is “nonsense,” since it is not defined.

Definition 5.5. Suppose $A = [a_{i,j}]$ and $B = [b_{i,j}]$ are both $m \times n$ matrices. Then their *sum* is the $m \times n$ matrix $A + B = [a_{i,j} + b_{i,j}]$. That is, addition is carried out element-wise.

Example 5.6. Suppose we have the following three matrices:

$$A = \begin{bmatrix} 3 & 4 & 6 \\ 5 & 9 & 12 \end{bmatrix} \quad B = \begin{bmatrix} \pi & -4 & -5 \\ 2 & 2 & -2 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then $A + B$ exists but neither $A + C$ nor $B + C$ is defined. The sum of A and B is

$$\begin{aligned} A + B &= \begin{bmatrix} 3 & 4 & 6 \\ 5 & 9 & 12 \end{bmatrix} + \begin{bmatrix} \pi & -4 & -5 \\ 2 & 2 & -2 \end{bmatrix} \\ &= \begin{bmatrix} 3 + \pi & 4 - 4 & 6 - 5 \\ 5 + 2 & 9 + 2 & 12 - 2 \end{bmatrix} \\ &= \begin{bmatrix} 3 + \pi & 0 & 1 \\ 7 & 11 & 10 \end{bmatrix}. \end{aligned}$$

<

Since we have the ability to index `other` when it is a `Matrix`, this should be straightforward to implement as a method. Moreover, matrix addition is *commutative*: if A and B are matrices such that $A + B$ exists, then $A + B = B + A$. This allows us to define `Matrix.__radd__` as a call to `Matrix.__add__` like we did in class with `ComplexNumber` in Lesson 4.

Exercise 5.7. Here's the code for matrix addition, with an important piece missing.

```
def __add__(self, other):
    if type(self) == type(other):
        if self.dims() != other.dims():
            raise ValueError('Cannot add matrices of different dimensions.')
        new_grid = "" "AN IMPORTANT PIECE IS MISSING RIGHT HERE!!" "" # <=====
        return Matrix(new_grid)

def __radd__(self, other):
    return self.__add__(other)
```

Correct the missing piece with a nested list comprehension. Put your corrected version into `Matrix.py`.

Exercise 5.8. Subtraction of matrices is a combination of addition and scalar multiplication: $A - B = A + (-1)B$. The builtin for subtraction using the “minus sign” is `--sub--`. Define `Matrix.sub` and add it to `Matrix.py`.

5.4 Matrix Multiplication

While multiplying two matrices together is not too much more complicated than either addition or scalar multiplication, we need to be very careful. We also will get to introduce another builtin command, `sum`.

Definition 5.9. Suppose A is a $m \times n$ matrix and B is a $p \times q$ matrix. Then the *matrix product* AB is defined if and only if $n = p$. In this case, AB is a $m \times q$ matrix given by

$$AB = \left[\sum_{k=0}^n a_{i,k} b_{k,j} \right].$$

That is, the element in the i^{th} row and j^{th} column of the product matrix AB is the sum of the element-wise product of the i^{th} row of A and the j^{th} column of B .

Example 5.10. Suppose we have the following matrices:

$$A = \begin{bmatrix} 2 & 3 & \frac{1}{2} \\ 1 & -1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 3 & 0 & 1 \\ -1 & -1 & 0 \\ 2 & 4 & 6 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 4 \\ \pi & 6 \\ 2 & 1 \end{bmatrix}.$$

Then A is a 2×3 matrix, B is a 3×3 matrix, and C is a 3×2 matrix. Here are the possible products, and if they exist, the dimensions of the product matrix:

Product	Exists?	Dimensions
AB	Yes	$(2 \times 3) \cdot (3 \times 3) \rightarrow 2 \times 3$
BA	No	$(3 \times 3) \cdot (2 \times 3) \rightarrow \emptyset$
AC	Yes	$(2 \times 3) \cdot (3 \times 2) \rightarrow 2 \times 2$
CA	Yes	$(3 \times 2) \cdot (2 \times 3) \rightarrow 3 \times 3$
BC	Yes	$(3 \times 3) \cdot (3 \times 2) \rightarrow 3 \times 2$
CB	No	$(3 \times 2) \cdot (3 \times 3) \rightarrow \emptyset$

Since the dimensions of AC and CA are different, we see that even if both AC and CA exist, they need not be equal! This is a simple demonstration of the fact that matrix multiplication is *not commutative*. So we can see the process of multiplying in action,

$$\begin{aligned} AC &= \begin{bmatrix} 2 & 3 & \frac{1}{2} \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ \pi & 6 \\ 2 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 2(1) + 3(\pi) + \frac{1}{2}(2) & 2(4) + 3(6) + \frac{1}{2}(1) \\ 1(1) - 1(\pi) + 0(2) & 1(4) - 1(6) + 0(1) \end{bmatrix} \\ &= \begin{bmatrix} 3 + 3\pi & \frac{53}{2} \\ 1 - \pi & -2 \end{bmatrix}, \end{aligned}$$

but in the other order

$$\begin{aligned} CA &= \begin{bmatrix} 1 & 4 \\ \pi & 6 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 & \frac{1}{2} \\ 1 & -1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1(2) + 4(1) & 1(3) + 4(-1) & 1(\frac{1}{2}) + 4(0) \\ \pi(2) + 6(1) & \pi(3) + 6(-1) & \pi(\frac{1}{2}) + 6(0) \\ 2(2) + 1(1) & 2(3) + 1(-1) & 2(\frac{1}{2}) + 1(0) \end{bmatrix} \\ &= \begin{bmatrix} 6 & -1 & \frac{9}{2} \\ 6 + 2\pi & -6 + 3\pi & \frac{9}{2} \\ 5 & 5 & 1 \end{bmatrix}. \end{aligned}$$

◁

To think about this algorithmically, let's say A is a $m \times n$ matrix and B is a $n \times q$ matrix. Then $C = AB$ is a $m \times q$ matrix; if $C = [c_{i,j}]$, then

$$c_{i,j} = \sum_{k=0}^n a_{i,k} b_{k,j}.$$

So for each pair i, j where i in `range(m)` and j in `range(q)` we will need to take the sum over a list `[A[i,k] * B[k,j] for k in range(n)]`. This is precisely the result of the `sum` command: it computes the sum of numbers in a list. The most efficient way to implement this is as a list comprehension of a list comprehension of a sum of a list comprehension! Here's an updated form of `Matrix.__mul__`, again with the list comprehension for `new_grid` missing.

```
def __mul__(self, other):
    mult_err = 'Matrices may only be multiplied by scalars, or '+\
               'by matrices of appropriate dimensions.'
    if type(self) == type(other): # matrix multiplication
        m,n = self.dims()
        p,q = other.dims()
        if n != p:
            raise ValueError(mult_err)
        new_grid = ""AN IMPORTANT PIECE IS MISSING RIGHT HERE!!"" # <=====
        return Matrix(new_grid)
    elif type(other) in [type(1), type(0.1)]: # scalar multiplication
        new_grid = ""AN IMPORTANT PIECE IS MISSING RIGHT HERE!!"" # <=====
        return Matrix(new_grid)
    else:
        raise TypeError(mult_err)
```

Exercise 5.11. *Finish the implementation of `Matrix.__mul__` to enable matrix multiplication. Save it in `Matrix.py`.*

Lesson 6

Vectors and solving systems of equations

Goals

1. Define a vector, mathematically.
2. Discuss the algebraic operations on vectors.
3. Discuss algorithms for basic vector operations.
4. Implement basic vector operations as a Python subclass.
5. Discuss solutions to systems of linear equations.
6. Introduce Gaussian elimination and Gauss-Jordan elimination.
7. Discuss pivoting strategies and their necessity.
8. Implement elementary row operations and discuss the algorithm for Gaussian elimination with partial pivoting.

6.1 Vectors in Euclidean space

The set of all real numbers, \mathbb{R} , can be considered (geometrically) as a Euclidean line – a one-dimensional space. Likewise, we are familiar with the way in which the set of all ordered pairs of real numbers represents a Euclidean plane, which we write as $\mathbb{R}^2 = \{(x, y) : x, y \in \mathbb{R}\}$. Extending this idea we can think of Euclidean 3-space, as $\mathbb{R}^3 = \{(x, y, z) : x, y, z \in \mathbb{R}\}$. Under a right-hand rule, this is geometrically shown in Figure 6.1. We can extend this idea to that of *Euclidean n -space*, given by

$$\mathbb{R}^n = \{(x_1, x_2, \dots, x_n) : x_i \in \mathbb{R} \text{ for all } i = 1, 2, \dots, n\},$$

although we (as limited humans in a 3-dimensional spatial universe) cannot picture higher dimensions. Points in n -dimensional Euclidean space are *n -dimensional vectors over \mathbb{R}* .

There is much disagreement among textbooks as to the correct notation for vectors, so in the interest of keeping everyone on the same page we will use the same notation used in Stewart's Calculus series: when written in *coordinate notation*, we will write $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$.

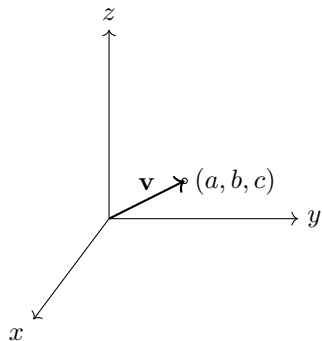


Figure 6.1: A drawing of a vector in 3-space.

As we will frequently combine matrix and vector calculations (for reasons which will become obvious), we will most often use *column vector* notation:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

6.2 Operations on vectors

Since one of the notations for a vector \mathbf{x} is as a column matrix, the operations for vectors must agree with those for matrices. Hence addition and scalar multiplication are carried out coordinate-wise.

Definition 6.1. Let $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$ and $\mathbf{y} = \langle y_1, y_2, \dots, y_n \rangle$, and let $k \in \mathbb{R}$. Then the scalar multiple of \mathbf{x} by k is the vector

$$k\mathbf{x} = \langle kx_1, kx_2, \dots, kx_n \rangle = \mathbf{x}k.$$

Likewise, the vector sum of \mathbf{x} and \mathbf{y} is the vector

$$\mathbf{x} + \mathbf{y} = \langle x_1 + y_1, x_2 + y_2, \dots, x_n + y_n \rangle = \mathbf{y} + \mathbf{x}.$$

Combining these defines vector subtraction,

$$\mathbf{x} - \mathbf{y} = \mathbf{x} + (-1)\mathbf{y}.$$

Rather than having exactly one way to multiply vectors, there could be many; we'll discuss the most generally useful, the *dot product*.

Definition 6.2. The *dot product* of vectors $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$ and $\mathbf{y} = \langle y_1, y_2, \dots, y_n \rangle$ is the scalar quantity

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

The *magnitude* of \mathbf{x} is defined to be

$$\|\mathbf{x}\| = \sqrt{\mathbf{x} \cdot \mathbf{x}} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

A final concept which is very important is the idea of multiplying a vector by a matrix. In a very literal sense, multiplying a vector on the left by a matrix of the correct dimensions is a special type of function on the vector space: a *linear transformation*.

Definition 6.3. Suppose $A = [a_{i,j}]$ is a real $m \times n$ matrix and $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$ is an n -dimensional real vector. Then $A\mathbf{x}$ is the m -dimensional vector whose i^{th} component is $\sum_{k=1}^n a_{i,k}x_k$.

$$A\mathbf{x} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \end{bmatrix}.$$

In this case, left-multiplication by A is a *linear transformation* mapping R^n to R^m . The study of vector spaces and linear transformations is called *linear algebra*.

For all of these operations, as well as instantiation and indexing, it is straightforward to develop Python code for a `Vector` class from the definitions; it is normal to consider vectors as immutable and hence to store the coordinates in a `tuple`. The important methods to define are `__init__`, `__repr__`, `dim` to return the dimension of the vector, `__getitem__` for access to the vector components, `__add__`, `__mul__` for scalar multiplication only, `dot` for dot products, and `norm` for the magnitude. A good place to write your `Vector` class is at the end of your `Matrix.py` file. This is especially important as implementing matrix-vector multiplication requires some careful consideration.

6.3 Motivation for matrix arithmetic

Matrices have only been called such¹ since 1850, but some of the methods for which they have been used have been known in varying parts of the world since as early as the second century BC. Nearly all earliest uses of matrices are for the same purpose: solving systems of simultaneous linear equations.

Example 6.4. Consider the following system of simultaneous linear equations in three variables:

$$\begin{cases} x_1 + 3x_2 - 13x_3 = -18 \\ x_1 + 2x_2 - 10x_3 = -15 \\ -6x_1 - 7x_2 + 46x_3 = 77 \end{cases} \quad (6.1)$$

We can solve this by the method of elimination, using repeated applications of the following three operations:

¹Matrix is latin for “womb,” and here’s the explanation for the term from the person who coined it, James Joseph Sylvester: *I have in previous papers defined a “Matrix” as a rectangular array of terms, out of which different systems of determinants may be engendered as from the womb of a common parent.*

1. The order of equations may be rearranged. For example, the first and third equation could be swapped denoted $E_1 : E_3$.
2. An equation may be multiplied by a nonzero constant. For example, the second equation could be multiplied by 6, denoted $6E_2$.
3. A nonzero multiple of one equation may be added to another equation. For instance, 3 times the first equation could be added to the second, denoted $3E_1 + E_2$

As these operations preserve the arithmetic properties of the equations, the set of solutions before and after any sequence of these operations must be the same. The goal in the method of elimination is to “diagonalize” the equation, so that the coefficient of x_j in row i is 0 for all $j < i$. For instance, with our example, we could proceed in the following order:

$$\begin{aligned} \left\{ \begin{array}{l} x_1 + 3x_2 - 13x_3 = -18 \\ x_1 + 2x_2 - 10x_3 = -15 \\ -6x_1 - 7x_2 + 46x_3 = 77 \end{array} \right. & \xrightarrow[6E_1+E_3]{-E_1+E_2} \left\{ \begin{array}{l} x_1 + 3x_2 - 13x_3 = -18 \\ -x_2 + 3x_3 = 3 \\ 11x_2 - 32x_3 = -31 \end{array} \right. \\ & \xrightarrow{11E_2+E_3} \left\{ \begin{array}{l} x_1 + 3x_2 - 13x_3 = -18 \\ -x_2 + 3x_3 = 3 \\ x_3 = 2 \end{array} \right. \end{aligned} \quad (6.2)$$

So eliminated, we can use back substitution to solve for x_3 , x_2 , and x_1 in that order:

$$\begin{aligned} x_3 &= 2 \\ x_2 &= \frac{1}{3}(3 + x_3) = 3 \\ x_1 &= -18 - 3x_2 + 13x_3 = -1 \end{aligned}$$

The importance of this method cannot be too highly stressed: solving systems of linear equations is a constant problem in applied mathematics, often because the underlying systems of nonlinear equations can be nicely linearized by making acceptable sacrifices. We notice that at no step were we required to interchange the order of equations, as we never encountered a situation where the i^{th} variable had a coefficient of 0 in the i^{th} row. \triangleleft

The process used in the preceding example has nothing to do with the variables used – in fact, they are used solely as placeholders in the computation. Understanding this, we can recast the problem into a vector algebra problem and dispense with the variables entirely.

Example 6.5. Consider the matrix A and vectors \mathbf{x} and \mathbf{b} given by

$$A = \begin{bmatrix} 1 & 3 & -13 \\ 1 & 2 & -10 \\ -6 & -7 & 46 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -18 \\ -15 \\ 77 \end{bmatrix}.$$

Then the system in Equation 6.1 is exactly equivalent to the vector equation $A\mathbf{x} = \mathbf{b}$. In order to keep track of the operations performed on both the left and right of the equal sign,

it is sufficient to *augment* the matrix A by the vector \mathbf{b} , like so:

$$A|\mathbf{b} = \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 1 & 2 & -10 & -15 \\ -6 & -7 & 46 & 77 \end{array} \right]. \quad (6.3)$$

Now the three operations of the method of elimination correspond to the *elementary row operations* on an augmented matrix, and the operations involved in Equation 6.2 correspond to the following sequence of row operations:

$$\begin{aligned} \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 1 & 2 & -10 & -15 \\ -6 & -7 & 46 & 77 \end{array} \right] &\xrightarrow[6R_1+R_3]{-R_1+R_2} \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 0 & -1 & 3 & 3 \\ 0 & 11 & -32 & -31 \end{array} \right] \\ &\xrightarrow{11R_2+R_3} \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 0 & -1 & 3 & 3 \\ 0 & 0 & 1 & 2 \end{array} \right] \end{aligned} \quad (6.4)$$

◁

This is a *row echelon form* for the matrix $A|\mathbf{b}$, and the process of obtaining it is called *Gaussian elimination*, or informally *row reduction*. If rows are interchanged or scaled, there can be many distinct row echelon forms of a matrix. However, we could continue to perform row operations until the left-most nonzero entry in each row is a 1 and it is the only non-zero element in its column. This extended method is called *Gauss-Jordan elimination*.

Example 6.6. Continuing from Equation 6.4, here are the final transformations from row echelon form to reduced row echelon form via Gauss-Jordan elimination.

$$\begin{aligned} \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 0 & -1 & 3 & 3 \\ 0 & 0 & 1 & 2 \end{array} \right] &\xrightarrow{-R_2} \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 0 & 1 & -3 & -3 \\ 0 & 0 & 1 & 2 \end{array} \right] \xrightarrow{-3R_2+R_1} \left[\begin{array}{ccc|c} 1 & 0 & -4 & -9 \\ 0 & 1 & -3 & -3 \\ 0 & 0 & 1 & 2 \end{array} \right] \\ &\xrightarrow[3R_3+R_2]{4R_3+R_1} \left[\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \end{array} \right] \end{aligned}$$

Analytically, there is no difference between Gauss-Jordan elimination and regular Gaussian elimination to row echelon form followed by back substitution. However, the computational complexity² of Gauss-Jordan is higher than elimination and substitution.

Gauss-Jordan elimination becomes necessary when we desire to compute the *inverse* of a nonsingular square matrix, later in the course. ◁

²Essentially, this is a measurement of the number of operations performed by an algorithm.

6.4 A floating-point problem

According to the specification for floating point arithmetic³, “double-precision” floating point numbers take up 64 bits of memory and are stored in the following manner:

$$f = (-1)^s \cdot c \cdot b^{(-1)^{q_s} q},$$

where

1. s is either 0 (for positive numbers) or 1 (for negatives),
2. c is an integer taking up 52 binary digits (bits),
3. q_s is the sign of the exponent, either 0 or 1, and
4. q is an integer taking up 10 bits.

The important part here is this: since c takes up 52 bits and is an integer, it can be any number from 0 to $2^{52} - 1 = 4,503,599,627,370,495$. What this means is that double-precision arithmetic is only precise to 16 decimal places; beyond that, no differences are noticed by the representation. This causes a very specific kind of problem, which we’ll call *swamping*. This won’t occur in our previous example, so we’ll discuss a new one.

Swamping

Consider the following matrix equation:

$$\begin{bmatrix} 10^{-20} & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}.$$

If we solve via Gaussian elimination without interchanging rows we’re going to have a problem almost immediately.

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 2 & 4 \end{array} \right] \xrightarrow{-10^{20} R_1 + R_2} \left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & 2 - 10^{20} & 4 - 10^{20} \end{array} \right]$$

No problem, right? Converting back to the original equation format, we see use back-substitution and find

$$\begin{aligned} x_2 &= \frac{4 - 10^{20}}{2 - 10^{20}} \approx 1.0, \\ x_1 &= 10^{20}(1 - x_2) = \frac{-2 \cdot 10^{20}}{2 - 10^{20}} \approx 2.0. \end{aligned}$$

Except there’s a problem: 10^{20} has so many digits that when represented as floating point numbers,

$$2.0 - 10.0^{20} = -10.0^{20} = 4.0 - 10.0^{20}.$$

³IEEE 754

Hence in floating point arithmetic, what we obtain is different:

$$\left[\begin{array}{cc|c} 10.0^{-20} & 1.0 & 1.0 \\ 1.0 & 2.0 & 4.0 \end{array} \right] \xrightarrow{-10.0^{20} R_1 + R_2} \left[\begin{array}{cc|c} 10.0^{-20} & 1.0 & 1.0 \\ -0.0 & -10.0^{20} & -10.0^{20} \end{array} \right]$$

Back substituting, again in floating point arithmetic, we have

$$x_2 = \frac{-10.0^{20}}{-10.0^{20}} = 1.0$$

$$x_1 = 10.0^{20}(1.0 - x_2) = 0.0$$

This is far from correct. In fact,

$$\begin{bmatrix} 10^{-20} & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

Thankfully there are solutions to these sorts of problems, called *pivoting strategies*.

6.5 Pivoting strategies

A *pivoting strategy* is an algorithm which decides when to interchange rows (and/or columns) to avoid errors such as swamping.

Partial pivoting

In partial pivoting, we consider the magnitude of elements in the column before selecting a pivot element. The unpivoted row containing an element of maximum magnitude in a column is first swapped with the pivot row, and then the elimination step for that row proceeds. Consider the augmented matrix in Equation 6.3 once again. In each step, the column entry of maximum magnitude is underlined for easy location, and then (if necessary) swapped into the correct location. Once the “largest” element is in the pivot position, we’ll use the elimination step of sending all column values beneath the pivot to 0.

$$\begin{aligned} \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 1 & 2 & -10 & -15 \\ \underline{-6} & -7 & 46 & 77 \end{array} \right] & \xrightarrow{R_1:R_3} \left[\begin{array}{ccc|c} -6 & -7 & 46 & 77 \\ 1 & 2 & -10 & -15 \\ 1 & 3 & -13 & -18 \end{array} \right] \xrightarrow{\begin{array}{l} \frac{1}{6} R_1 + R_2 \\ \frac{1}{6} R_1 + R_3 \end{array}} \left[\begin{array}{ccc|c} -6 & -7 & 46 & 77 \\ 0 & \frac{5}{6} & -\frac{7}{3} & -\frac{13}{6} \\ 0 & \underline{\frac{11}{6}} & -\frac{16}{3} & -\frac{31}{6} \end{array} \right] \\ & \xrightarrow{R_2:R_3} \left[\begin{array}{ccc|c} -6 & -7 & 46 & 77 \\ 0 & \frac{11}{6} & -\frac{16}{3} & -\frac{31}{6} \\ 0 & \frac{5}{6} & -\frac{7}{3} & -\frac{13}{6} \end{array} \right] \xrightarrow{-\frac{5}{11} R_2 + R_3} \left[\begin{array}{ccc|c} -6 & -7 & 46 & 77 \\ 0 & \frac{11}{6} & -\frac{16}{3} & -\frac{31}{6} \\ 0 & 0 & \frac{1}{11} & \frac{2}{11} \end{array} \right] \end{aligned}$$

Following up with back substitution,

$$x_3 = \frac{2/11}{1/11} = 2,$$

$$x_2 = \frac{6}{11} \left(-\frac{31}{6} + \frac{16}{3} x_3 \right) = 3,$$

$$x_1 = -\frac{1}{6} (77 + 7x_2 - 46x_3) = -1$$

In this example, we see no net gain; in the example with swamping, we will.

Example 6.7. Recall our swamping example. This time we'll use floating-point arithmetic but include partial pivoting.

$$\left[\begin{array}{cc|c} 10.0^{-20} & 1.0 & 1.0 \\ 1.0 & 2.0 & 4.0 \end{array} \right] \xrightarrow{R_1 \leftrightarrow R_2} \left[\begin{array}{cc|c} 1.0 & 2.0 & 4.0 \\ 10.0^{-20} & 1.0 & 1.0 \end{array} \right] \xrightarrow{-10.0^{-20} R_1 + R_2} \left[\begin{array}{cc|c} 1.0 & 2.0 & 4.0 \\ 0.0 & 1.0 & 1.0 \end{array} \right]$$

Back substitution is immediate:

$$\begin{aligned} x_2 &= 1.0, \\ x_1 &= 4.0 - 2.0x_2 = 2.0. \end{aligned}$$

So we see that using partial pivoting totally avoided the error. The small difference which occurred and was ignored did so in the correct level of significance, and the actual difference between the correct answer and its floating point approximation is below the level of precision of floating point arithmetic! \triangleleft

Complete Pivoting

Complete pivoting involves swapping the columns as well as the rows, which is slightly dangerous – this changes the order of the solutions. An interesting way to track these changes is to use a *permutation matrix*. Suppose that A is a $m \times n$ matrix, and I is the $n \times n$ identity matrix. Then if A' and I' are the results of swapping the i^{th} and j^{th} columns in both A and I , it is the case that

$$A = AI = A'I'.$$

Hence a sequence of column swaps on A can be tracked by making the same sequence of column swaps on an identity matrix I of the correct dimensions.

The pivot selection for complete pivoting is to swap the largest-magnitude entry in the submatrix below and to the right of the previous pivot point into the current pivot position via a row and column swap.

6.6 Algorithms for Gaussian elimination

Beginning with implementations of the elementary row operations, we can build the full algorithm for the method of Gaussian elimination with partial pivoting.

Elementary row operation: Elimination

The first elementary row operation is the elimination operation. We write this in our shorthand as $kR_i + R_j$ and mean “multiply every element of row i by scalar k and then add that product to the corresponding element of row j .” Rather than trying to perform the matrix operations as changes to a matrix itself, we'll assume that a new matrix is to be returned.

Algorithm 6.8. Suppose $A = [a_{i,j}]$ is a Matrix, k is a scalar (float), and i and j are row indices.

1. Let `temp` be a list of the tuples in A .
2. Assign `temp[j]` to be the list containing $ka_{i,\ell} + a_{j,\ell}$ for each ℓ .
3. Return the Matrix of `temp`.

Elementary row operation: Swapping rows

The next elementary row operation is changing the row order⁴ and is used in the partial pivoting algorithm.

Algorithm 6.9. Suppose $A = [a_{i,j}]$ is a Matrix and both i and j are row indices.

1. Let `temp` be a list of the tuples of A .
2. Set `temp[i]` to the j^{th} tuple of A .
3. Set `temp[j]` to the i^{th} tuple of A .
4. Return the Matrix of `temp`.

Gaussian elimination with partial pivoting

While we could first implement so-called naive Gaussian elimination, it makes more sense to immediately write the algorithm for partial pivoting as well.

Algorithm 6.10. Suppose A is a $m \times n$ Matrix. Set $i = 0$ and $j = 0$.

1. (Pivot step) In the j^{th} column, identify a row containing an element of maximum absolute value. Let the index of this row be p .
 - a. If that value is 0, increment j . If after incrementing, $j > n$, return the matrix.
 - b. Otherwise, swap row i and row p .
2. For every row index p greater than i , eliminate the coefficient in $A[p, j]$ if it is not already 0.
3. Increment i and j . If after incrementing either $i > m$ or $j > n$, return the matrix.

At this point, you should be able to discover and implement the algorithm for back-substitution by yourself. What does it mean if there is a column of an augmented matrix which is never used for elimination?

⁴Even though we only swap two rows at a time, this is still a *permutation of the rows*. We'll talk about permutations a lot when we get to Cryptography.

Lesson 7

Matrix decomposition: LU decomposition

Goals

1. Discuss elementary matrices and their relationship to Gaussian elimination
2. Discuss matrix inverses
3. Discuss the LU decomposition of an invertible matrix A
4. Discuss the $PA = LU$ factorization of an invertible matrix A

Remark. We often talk about square matrices in the context of systems of linear equations because a system of fewer than n equations in n unknowns is *underqualified*; that is, there are too few equations to give a unique solution in all variables. On the other hand, systems of more than n equations in n variables either have redundant equations (at least one equations is a linear combination of the others) or the system is *overqualified* and an inconsistent system. Thus n equations in n unknowns is the sort of “happy medium” of having enough equations for unique solutions without having too many for a solution to exist.

7.1 Elementary matrices

Suppose there is a matrix A , and some sequence of row operations transforms it into A' , so that $A \rightarrow A'$. What happens if we assume that there is some matrix E such that $EA = A'$? Assuming that such an E exists, what can we say about it?

It makes sense to think that nothing is special about the matrix A when performing a row operation; for instance, $-2R_1 + R_3$ always produces elements $a_{3,j} - 2a_{1,j}$ in the third row, no matter what the values of $a_{i,j}$ are. So we should think that if E corresponds to a particular elementary row operation, then left-multiplying *any* matrix of the correct dimensions by E will have the proper result. Further, $EI = E$, so we know that whatever effect a row operation has on the identity matrix results in the elementary matrix itself!

Row swaps

Consider then the result on the 4×4 identity matrix of having rows 1 and 3 swapped.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{R_1:R_3} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Hence the matrix corresponding to interchanging rows k and ℓ must be the matrix with 0s in all entries except for 1s when $i = j$ and $i \neq k, i \neq \ell$, and $a_{k,\ell} = a_{\ell,k} = 1$. That is, $E = [e_{i,j}]$ with

$$e_{i,j} = \begin{cases} 1, & i = j \text{ and } i, j \notin \{k, \ell\}, \text{ or } i \neq j \text{ and } i, j \in \{k, \ell\} \\ 0, & \text{otherwise.} \end{cases}$$

Scalar multiplication of a row

Again, operating on the 4×4 identity matrix gives insight to the structure of this elementary matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{2R_4} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}.$$

Hence the $n \times n$ matrix which corresponds to multiplying row k by a scalar λ is the matrix $E = [e_{i,j}]$ where

$$e_{i,j} = \begin{cases} 1, & i = j \text{ and } i \neq k \\ \lambda, & i = j = k \\ 0, & \text{otherwise.} \end{cases}$$

Adding a scalar multiple of one row to another row

Once again, consider the 4×4 identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{9R_4+R_2} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Hence the $n \times n$ matrix which corresponds to adding to each column of row ℓ the scalar multiple λ times the corresponding column of row k ($\lambda R_k + R_\ell$) is the matrix $E = [e_{i,j}]$ where

$$e_{i,j} = \begin{cases} 1, & i = j \\ \lambda, & i = \ell \text{ and } j = k \\ 0, & \text{otherwise.} \end{cases}$$

Note there that the value λ appears in the row being added to and in the column corresponding to the row which is being multiplied.

Theorem 7.1. *Suppose that A is a $n \times n$ matrix and that there is some sequence of elementary row operations by which A can be transformed into I , the $n \times n$ identity matrix. Then there is a sequence (E_1, E_2, \dots, E_k) of elementary matrices such that*

$$E_k E_{k-1} \cdots E_2 E_1 A = I.$$

□

7.2 Matrix inverses

Definition 7.2. Let A be an $n \times n$ matrix. Suppose there is a matrix B such that $AB = I$. Then B is the *inverse* of A , and we write A^{-1} instead of B . Moreover, A^{-1} is the unique such matrix and $A^{-1}A = I = AA^{-1}$.

Theorem 7.3. *Suppose A is a $n \times n$ matrix and there is a sequence (E_1, E_2, \dots, E_k) of elementary matrices such that*

$$E_k E_{k-1} \cdots E_2 E_1 A = I.$$

Then $A^{-1} = E_k E_{k-1} \cdots E_2 E_1$.

Proof. From the definition and Theorem 7.1. □

Application of Gauss-Jordan elimination

We find the inverse of a matrix by the same process of Gauss-Jordan elimination as used when solving systems of equations, but now we augment our square matrix A by an appropriately sized identity matrix rather than by a single vector.

Example 7.4. Suppose we begin with a 3×3 matrix augmented by I

$$A = \left[\begin{array}{ccc|ccc} 1 & 3 & 5 & 1 & 0 & 0 \\ 7 & 9 & 11 & 0 & 1 & 0 \\ 2 & -4 & -6 & 0 & 0 & 1 \end{array} \right].$$

Just to demonstrate that the choice of elementary row operations does not matter, we'll use Gauss-Jordan elimination with partial pivoting. Row operations marked on arrows are performed in order from the top to the bottom.

$$\begin{aligned}
 \left[\begin{array}{ccc|ccc} 1 & 3 & 5 & 1 & 0 & 0 \\ 7 & 9 & 11 & 0 & 1 & 0 \\ 2 & -4 & -6 & 0 & 0 & 1 \end{array} \right] & \xrightarrow[\begin{smallmatrix} -R_1+R_2 \\ -2R_1+R_3 \end{smallmatrix}]{\begin{smallmatrix} R_1:R_2 \\ \frac{1}{7}R_1 \end{smallmatrix}} \left[\begin{array}{ccc|ccc} 1 & \frac{9}{7} & \frac{11}{7} & 0 & \frac{1}{7} & 0 \\ 0 & \frac{12}{7} & \frac{24}{7} & 1 & -\frac{1}{7} & 0 \\ 0 & -\frac{46}{7} & -\frac{64}{7} & 0 & -\frac{2}{7} & 1 \end{array} \right] \\
 & \xrightarrow[\begin{smallmatrix} -\frac{9}{7}R_2+R_1 \\ -\frac{12}{7}R_2+R_3 \end{smallmatrix}]{\begin{smallmatrix} R_2:R_3 \\ -\frac{7}{46}R_2 \end{smallmatrix}} \left[\begin{array}{ccc|ccc} 1 & 0 & -\frac{5}{23} & 0 & \frac{2}{23} & \frac{9}{46} \\ 0 & 1 & \frac{32}{23} & 0 & \frac{1}{23} & -\frac{7}{46} \\ 0 & 0 & \frac{24}{23} & 1 & -\frac{5}{23} & \frac{6}{23} \end{array} \right] \\
 & \xrightarrow[\begin{smallmatrix} \frac{5}{23}R_3+R_1 \\ -\frac{32}{23}R_3+R_2 \end{smallmatrix}]{\frac{23}{24}R_3} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{5}{24} & \frac{1}{24} & \frac{1}{4} \\ 0 & 1 & 0 & -\frac{4}{3} & \frac{1}{3} & -\frac{1}{2} \\ 0 & 0 & 1 & \frac{23}{24} & -\frac{5}{24} & \frac{1}{4} \end{array} \right]
 \end{aligned}$$

Now, observe:

$$\begin{aligned}
 \left[\begin{array}{ccc} 1 & 3 & 5 \\ 7 & 9 & 11 \\ 2 & -4 & 6 \end{array} \right] \left[\begin{array}{ccc} \frac{5}{24} & \frac{1}{24} & \frac{1}{4} \\ -\frac{4}{3} & \frac{1}{3} & -\frac{1}{2} \\ \frac{23}{24} & -\frac{5}{24} & \frac{1}{4} \end{array} \right] &= \left[\begin{array}{ccc} 1(\frac{5}{24})+3(-\frac{4}{3})+5(\frac{23}{24}) & 1(\frac{1}{24})+3(\frac{1}{3})+5(-\frac{5}{24}) & 1(\frac{1}{4})+3(-\frac{1}{2})+5(\frac{1}{4}) \\ 7(\frac{5}{24})+9(-\frac{4}{3})+11(\frac{23}{24}) & 7(\frac{1}{24})+9(\frac{1}{3})+11(-\frac{5}{24}) & 7(\frac{1}{4})+9(-\frac{1}{2})+11(\frac{1}{4}) \\ 2(\frac{5}{24})-4(-\frac{4}{3})-6(\frac{23}{24}) & 2(\frac{1}{24})-4(\frac{1}{3})-6(-\frac{5}{24}) & 2(\frac{1}{4})-4(-\frac{1}{2})-6(\frac{1}{4}) \end{array} \right] \\
 &= \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right]
 \end{aligned}$$

◁

7.3 LU decomposition

Definition 7.5. A $m \times n$ matrix L is *lower triangular* if its entries satisfy $\ell_{i,j} = 0$ for $j > i$ (entries to the right of the “main diagonal” are zero). A $m \times n$ matrix U is *upper triangular* if its entries satisfy $u_{i,j} = 0$ for $j < i$ (entries to the left of the “main diagonal” are zero). An LU decomposition of a matrix A is any factorization $A = LU$ where L is lower triangular and U is an upper triangular row echelon form of A .

We can apply our theory of elementary matrices to this problem, immediately!

Example 7.6. Recall from Lesson 6 our example system of equations and its corresponding augmented matrix form:

$$\begin{cases} x_1 + 3x_2 - 13x_3 = -18 \\ x_1 + 2x_2 - 10x_3 = -15 \\ -6x_1 - 7x_2 + 46x_3 = 77 \end{cases} \quad A|\mathbf{b} = \left[\begin{array}{ccc|c} 1 & 3 & -13 & -18 \\ 1 & 2 & -10 & -15 \\ -6 & -7 & 46 & 77 \end{array} \right].$$

The three required steps were $\xrightarrow{-R_1+R_2}$, $\xrightarrow{6R_1+R_3}$, and $\xrightarrow{11R_2+R_3}$, resulting in

$$U = \left[\begin{array}{ccc} 1 & 3 & -13 \\ 0 & -1 & 3 \\ 0 & 0 & 1 \end{array} \right]$$

when applied to just A . So if we let those correspond to elementary matrices E_1 , E_2 , and E_3 respectively, we get

$$E_3 E_2 E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 6 & 11 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & -13 \\ 1 & 2 & -10 \\ -6 & -7 & 46 \end{bmatrix} = \begin{bmatrix} 1 & 3 & -13 \\ 0 & -1 & 3 \\ 0 & 0 & 1 \end{bmatrix} = U.$$

If we left-multiply both sides of this equation with the correct inverse matrices and obtain a lower triangular matrix as their product, we'll be done! \triangleleft

At this point it becomes helpful to assign special letters to the elementary matrices; these are by no means standard throughout the linear algebra literature and serve mostly to make clear the following theorem. Generally, when you discuss elementary matrices, they will simply be E_i s.

Theorem 7.7. *All elementary matrices are invertible.*

1. If $S_{k,\ell}$ is an elementary matrix which swaps rows k and ℓ ($R_k : R_\ell$), then $S_{k,\ell}^2 = I$, and $S_{k,\ell} = S_{k,\ell}^{-1}$.
2. If $M_{\lambda,k}$ is an elementary matrix which multiplies row k by a scalar λ (λR_k), then $M_{\lambda,k}^{-1} = M_{1/\lambda,k}$.
3. If $E_{\lambda,k,j}$ is an elementary matrix which multiplies row k by a scalar λ and then adds the result onto row ℓ ($\lambda R_k + R_\ell$), then $E_{\lambda,k,j}^{-1} = E_{-\lambda,k,j}$.

Example 7.8. Continuing from above, we have

$$E_3 E_2 E_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 6 & 11 & 1 \end{bmatrix},$$

and all of E_1, E_2 , and E_3 are matrices as in Theorem 7.7 part 3. Hence

$$(E_3 E_2 E_1)^{-1} = E_1^{-1} E_2^{-1} E_3^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -6 & -11 & 1 \end{bmatrix}.$$

Since this is lower triangular, we say $L = E_1^{-1} E_2^{-1} E_3^{-1}$ and have obtained $A = LU$, the LU factorization of our matrix A . Now we need to use the LU decomposition to solve the system of equations! \triangleleft

Matrix algebra very quickly gives us a method of solving the matrix equation $A\mathbf{x} = \mathbf{b}$ using the LU decomposition:

$$\mathbf{b} = A\mathbf{x} = (LU)\mathbf{x} = L(U\mathbf{x}).$$

So we now can separate this into two problems, namely $L\mathbf{y} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$, but these are both triangular matrices. The solution to a this type of problem with triangular matrices involves only substitutions (either forward or backward)!

Example 7.9. Let's take our example and finish it out with substitution.

$$A = \begin{bmatrix} 1 & 3 & -13 \\ 1 & 2 & -10 \\ -6 & -7 & 46 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -6 & -11 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 1 & 3 & -13 \\ 0 & -1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

Recall that we must solve $L\mathbf{y} = \mathbf{b}$ before we can solve $U\mathbf{x} = \mathbf{y}$, and so doing we obtain

$$\begin{aligned} y_1 &= b_1 & x_3 &= y_3 \\ y_2 &= b_2 - y_1 & x_2 &= -(y_2 - 3x_3) \\ y_3 &= b_3 + 6y_1 + 11y_2 & x_1 &= y_1 - 3x_2 + 13x_3 \end{aligned}$$

As you can see, we can now solve the matrix equation $A\mathbf{x} = \mathbf{b}$ very quickly for any $\mathbf{b} = \langle b_1, b_2, b_3 \rangle$. In our example, we had $\mathbf{b} = \langle -18, -15, 77 \rangle$, so we get

$$\begin{aligned} y_1 &= b_1 = -18 & x_3 &= y_3 = 2 \\ y_2 &= b_2 - y_1 = 3 & x_2 &= -(y_2 - 3x_3) = 3 \\ y_3 &= b_3 + 6y_1 + 11y_2 = 2 & x_1 &= y_1 - 3x_2 + 13x_3 = -1 \end{aligned}$$

and our solution vector is $\mathbf{x} = \langle 2, 3, -1 \rangle$. ◁

Motivation for using LU decomposition: operational complexity

We have so far avoided discussing algorithmic complexity, but it is no longer avoidable.

Definition 7.10. Suppose that an algorithm requires $f(n)$ operational steps to complete when the input has size n^1 . Suppose also that there is some function $F(n)$ such that

$$\lim_{n \rightarrow \infty} \frac{F(n)}{f(n)} = c$$

is constant. Then we say the algorithm has a *big- O complexity* of $F(n)$, written $O(F(n))^2$.

This applies to the problem of solving systems of equations like so: a very well implemented naive algorithm for producing the Gaussian elimination of a matrix requires $\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n$ divisions, multiplications, and addition/subtractions to perform the elimination step, and then an additional n^2 of these operations to perform the back substitution step. Hence the operational complexity of naive Gaussian elimination is $O(n^3)$, since

$$\lim_{n \rightarrow \infty} \frac{\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n}{n^3} = \frac{2}{3}.$$

What if instead of solving an equation $A\mathbf{x} = \mathbf{b}$, we actually have to solve a system

$$A\mathbf{x} = \mathbf{b}_1, \quad A\mathbf{x} = \mathbf{b}_2, \quad A\mathbf{x} = \mathbf{b}_3, \quad \dots \quad A\mathbf{x} = \mathbf{b}_n,$$

¹Both of these are very vague, because individual operations differ between algorithms, as does the measurement of size of input.

²While any function F will do, we try to find as simple a function as possible while keeping the limit positive.

where A is a $n \times n$ matrix? In such a case, since our vectors \mathbf{b}_i are involved in every step of each Gaussian elimination, solving all n problems becomes a $O(n^4)$ algorithm, which is much worse than $O(n^3)$.

On the other hand, the operational complexity of finding the LU decomposition of A is still $O(n^3)$, since we do all the same steps but have some additional assignments into the matrix L . Using the LU decomposition to solve $(LU)\mathbf{x} = \mathbf{b}_i$ requires $2n^2$ operations for the forward substitution involved in solving $L\mathbf{y} = \mathbf{b}_i$ and then the back substitution of solving $U\mathbf{x} = \mathbf{y}$. Using the LU decomposition to solve the system of matrix equations above then is still $O(n^3)$, since

$$\lim_{n \rightarrow \infty} \frac{n^3 + 2n^2(n)}{n^3} = 3.$$

To sum up, the LU decomposition provides no net gain when used to solve a single matrix equation, but when faced with a large system of matrix equations it provides an enormous benefit in terms of the complexity of the algorithm and the time it will require for a program to complete the process.

7.4 $PA = LU$ factorization

All of the above work is for naught if there is a zero pivot, of course, or computationally if due to swamping we encounter insurmountable errors. The solution, of course, is to use partial pivoting – but permuting the rows of the matrix introduces a complication into the process. The solution is to find the appropriate *permutation matrix* P so that $PA = LU$.

Definition 7.11. A matrix P is a *permutation matrix* if it is a product of elementary matrices of type $S_{k,\ell}$ only.

In the LU factorization, the matrix L is used as a place to “store” the row multipliers which are used in elimination; a more clever implementation of the algorithm allows those values to be stored in the matrix U below the main diagonal! This is useful as it allows the rows to be pivoted and the scalar multipliers to stay with their correct row if rows need to be swapped for partial pivoting. The row interchanges will be tracked in a permutation matrix P .

Example 7.12. We will decompose a different example than previously to demonstrate how the multipliers stored within the matrix move with their rows. Boxed entries are “zeroes,” which for sake of convenience will have the multipliers stored in them.

$$\begin{aligned}
 A = \begin{bmatrix} 4 & 9 & -11 \\ -2 & 3 & 5 \\ 7 & 1 & 1 \end{bmatrix} &\xrightarrow[R_1 \cdot R_3]{P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}} \begin{bmatrix} 7 & 1 & 1 \\ -2 & 3 & 5 \\ 4 & 9 & -11 \end{bmatrix} \xrightarrow[R_3 - \frac{4}{7}R_1]{R_2 - (-\frac{2}{7}R_1)} \begin{bmatrix} 7 & 1 & 1 \\ \boxed{-\frac{2}{7}} & \frac{23}{7} & \frac{37}{7} \\ \boxed{\frac{4}{7}} & \frac{59}{7} & -\frac{81}{7} \end{bmatrix} \\
 &\xrightarrow[R_2 \cdot R_3]{P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}} \begin{bmatrix} 7 & 1 & 1 \\ \boxed{\frac{4}{7}} & \frac{59}{7} & -\frac{81}{7} \\ \boxed{-\frac{2}{7}} & \frac{23}{7} & \frac{37}{7} \end{bmatrix} \xrightarrow{R_3 - \frac{23}{59}R_2} \begin{bmatrix} 7 & 1 & 1 \\ \boxed{\frac{4}{7}} & \frac{59}{7} & -\frac{81}{7} \\ \boxed{-\frac{2}{7}} & \boxed{\frac{23}{59}} & \boxed{\frac{578}{59}} \end{bmatrix}
 \end{aligned}$$

Pulling apart this very compact notation, we have

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad A = \begin{bmatrix} 4 & 9 & -11 \\ -2 & 3 & 5 \\ 7 & 1 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{4}{7} & 1 & 0 \\ -\frac{2}{7} & \frac{23}{59} & 1 \end{bmatrix} \quad U = \begin{bmatrix} 7 & 1 & 1 \\ 0 & \frac{23}{7} & -\frac{81}{7} \\ 0 & 0 & \frac{578}{59} \end{bmatrix}$$

and can very easily check

$$PA = \begin{bmatrix} 7 & 1 & 1 \\ 4 & 9 & -11 \\ -2 & 3 & 5 \end{bmatrix} = LU.$$

◁

Here's code that can be embedded into `Matrix.py` to implement the $PA = LU$ factorization of a matrix A . This does not use any of the elementary row operation methods previously developed

```
def palu(self):
    m,n = self.dims()
    if m!=n:
        raise ValueError('Nonsquare matrix.')
    elif self.det() == 0:
        raise ValueError('Singular matrix.')
    else:
        P = [[1 if i==j else 0 for j in range(n)] for i in range(m)]
        mat = [[self[i,j] for j in range(n)] for i in range(m)]
        for j in range(m):
            maxv = abs(mat[j][j])
            piv = j
            for i in range(j+1,m):
                if abs(mat[i][j]) > maxv:
                    maxv = mat[i][j]
                    piv = i
            if piv != j:
                mat = mat[:j] + [mat[piv]] + mat[j+1:piv] + [mat[j]] \
                    + mat[piv+1:]
                P = P[:j] + [P[piv]] + P[j+1:piv] + [P[j]] + P[piv+1:]
            for i in range(j+1,m):
                mult = Fraction(mat[i][j],mat[j][j])
                mat[i][j] = mult
                for c in range(j+1,n):
                    mat[i][c] += -mult*mat[j][c]
        L = Matrix([[1 if i==j else (0 if j>i else mat[i][j])
                    for j in range(n)] for i in range(m)])
        U = Matrix([[mat[i][j] if j>=i else 0 for j in range(n)]
                    for i in range(m)])
        return (Matrix(P),L,U)
```

Lesson 8

Matrix decomposition: QR decomposition

Goals

1. Discuss least squares regression
2. Discuss the Gram-Schmidt orthonormalization algorithm
3. Discuss QR decomposition using Householder reflectors

8.1 Least squares regression

The least squares problem goes back to the beginnings of matrix theory, and in fact to the very problems which Gauss and Legendre considered during the early 1800s. Consider the scatter plot in Figure 8.1.

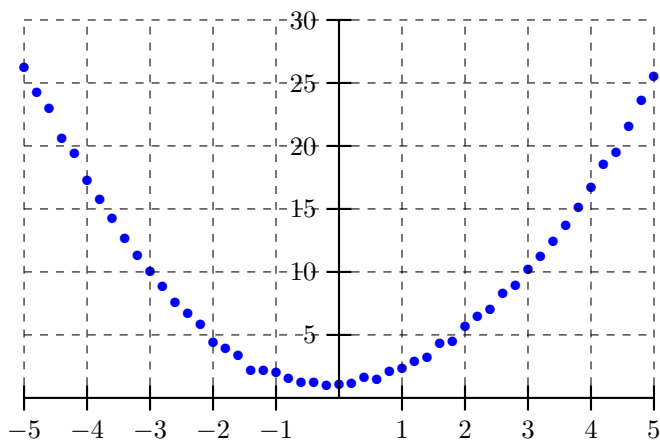


Figure 8.1: A scatter plot.

It appears that the points lie along some sort of parabola, but not exactly along the parabola – there seems to be some error involved in the fit of the curve. A question one might naturally ask is, “What is the equation of the parabola which best fits the given data?”

This and related questions are most often answered by the process of *least squares regression*. We’ll approach this problem by a seemingly unrelated means.

Inconsistent systems of equations

Consider the system

$$\begin{cases} 3x_1 - 2x_2 = 4 \\ 2x_1 + x_2 = 9 \\ -3x_1 + 2x_2 = 6. \end{cases} \quad (8.1)$$

It should be immediately clear that this is an inconsistent system, since not both the first and the third equations can be simultaneously valid. Hence there is no $\mathbf{x} = \langle x_1, x_2 \rangle$ which satisfies the system. Is it possible then to find some \bar{x} which is the “best” approximate solution?

We can recast the system Equation 8.1 as a vector and matrix problem (sadly which is still inconsistent) by writing it as $x_1\mathbf{v}_1 + x_2\mathbf{v}_2 = A\mathbf{x} = \mathbf{b}$ where

$$x_1\mathbf{v}_1 + x_2\mathbf{v}_2 = x_1 \begin{bmatrix} 3 \\ 2 \\ -3 \end{bmatrix} + x_2 \begin{bmatrix} -2 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 & -2 \\ 2 & 1 \\ -3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \\ 6 \end{bmatrix} = \mathbf{b}.$$

What’s interesting about this way of writing the system is that now the left-hand side is a *linear combination* of vectors \mathbf{v}_1 and \mathbf{v}_2 and the right-hand side is a vector \mathbf{b} which **can not** be written as a linear combination of \mathbf{v}_1 and \mathbf{v}_2 .

Definition 8.1. A vector \mathbf{w} is a *linear combination* from a set $S = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ if and only if there is a set of scalars $\{a_1, a_2, \dots, a_k\}$ such that

$$\mathbf{w} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \dots + a_k\mathbf{v}_k.$$

The set of all such vectors \mathbf{w} is the *span* of S . The set S is *linearly independent* if and only if no vector $\mathbf{v} \in S$ can be written as a linear combination of the vectors in $S \setminus \{\mathbf{v}\}$.

So now we can recast our problem in linear algebra terminology: we need to find the vector $\bar{x} \in \text{span}\{\mathbf{v}_1, \mathbf{v}_2\}$ which is nearest to \mathbf{b} . A convenient fact of working over the real numbers is that we already know how to do this: the shortest path from a line ℓ to a point P not on the line is the vector \mathbf{r} from $X \in \ell$ to P where the line \overline{XP} is perpendicular to ℓ . To extend this idea to higher dimensions, the vector \mathbf{r} must be *orthogonal* to any other vector \mathbf{v} in the linear subspace¹.

Definition 8.2. Two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ are *orthogonal* if and only if $\mathbf{u} \cdot \mathbf{v} = 0$.

¹Line, plane, or hyperplane!

An easy fact to prove is that the only vector in a space which is orthogonal to every vector in its space is the zero vector.

Theorem 8.3. Suppose $\mathbf{u}_0 \in \mathbb{R}^n$. If $\mathbf{u}_0 \cdot \mathbf{v} = 0$ for every $\mathbf{v} \in \mathbb{R}^n$, then $\mathbf{u}_0 = \mathbf{0}$.

Proof. Suppose $\mathbf{v} \in \mathbb{R}^n$ is an arbitrary vector. Then

$$\mathbf{0} \cdot \mathbf{v} = \sum_{i=1}^n 0v_i = 0,$$

so $\mathbf{0}$ is orthogonal to every vector in \mathbb{R}^n . Now suppose $\mathbf{u} \neq \mathbf{0}$; then at least one of the components of \mathbf{u} is nonzero; suppose that $u_k \neq 0$. Then

$$\mathbf{u} \cdot \mathbf{u} = \sum_{i=1}^n u_i^2 \geq u_k^2 > 0.$$

Hence \mathbf{u} is not orthogonal to itself, and therefore there is a nonzero vector in \mathbb{R}^n to which \mathbf{u} is not orthogonal. \square

The next theorem establishes that if $\mathbf{b} \notin \text{span}(S)$, then there is some vector in $\text{span}(S)$ which is the *closest approximation* of \mathbf{b} in $\text{span}(S)$. As the proof is more complicated, it is omitted.

Theorem 8.4. Suppose $S = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ is a set of linearly independent vectors in \mathbb{R}^n for $n > k$ and $\mathbf{b} \in \mathbb{R}^n \setminus \text{span}(S)$. Then there is a unique vector $\bar{\mathbf{u}} \in \text{span}(S)$ such that the magnitude of $\mathbf{r} = \mathbf{b} - \bar{\mathbf{u}}$ is minimum and \mathbf{r} is orthogonal to every vector $\mathbf{u} \in \text{span}(S)$.

Back to our problem, we're looking at vectors $\mathbf{v}_1 = \langle 3, 2, -3 \rangle$ and $\mathbf{v}_2 = \langle -2, 1, 2 \rangle$, and a vector $\mathbf{b} = \langle 4, 9, 6 \rangle \notin \text{span}\{\mathbf{v}_1, \mathbf{v}_2\}$. If we let

$$A = \begin{bmatrix} 3 & -2 \\ 2 & 1 \\ -3 & 2 \end{bmatrix},$$

we see that $\mathbf{u} \in \text{span}\{\mathbf{v}_1, \mathbf{v}_2\}$ if and only if there is some $\mathbf{x} = \langle x_1, x_2 \rangle \in \mathbb{R}^2$ such that $A\mathbf{x} = \mathbf{u}$. If we want to find $\bar{\mathbf{u}} = A\bar{\mathbf{x}}$ such that $\mathbf{r} = \mathbf{b} - A\bar{\mathbf{x}}$ is minimized, it suffices to find $\bar{\mathbf{x}}$ such that $(A\mathbf{x}) \cdot (\mathbf{b} - A\bar{\mathbf{x}}) = 0$ for all $\mathbf{x} \in \mathbb{R}^2$.

Lemma 8.5. If the vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ are considered as $n \times 1$ matrices, then $\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T \mathbf{v}$.

Supposing such a $\bar{\mathbf{x}}$ exists,

$$\begin{aligned} 0 &= (A\mathbf{x}) \cdot (\mathbf{b} - A\bar{\mathbf{x}}) = (A\mathbf{x})^T (\mathbf{b} - A\bar{\mathbf{x}}) \\ &= \mathbf{x}^T A^T (\mathbf{b} - A\bar{\mathbf{x}}) \end{aligned}$$

for all $\mathbf{x} \in \mathbb{R}^2$; but by Theorem 8.3 above, this means that for such a $\bar{\mathbf{x}}$ to exist it suffices that $A^T \mathbf{b} - A^T A \bar{\mathbf{x}} = \mathbf{0}$, or equivalently that $(A^T A) \bar{\mathbf{x}} = A^T \mathbf{b}$. This vector equation has a unique solution exactly when $A^T A$ can be reduced to the identity matrix via Gauss-Jordan elimination. The system of equations corresponding to the equation $(A^T A) \bar{\mathbf{x}} = A^T \mathbf{b}$ are called the *normal equations for least squares*.

Theorem 8.6 (Normal equations for least squares). *Suppose A is a $m \times n$ real matrix and $\mathbf{b} \in \mathbb{R}^m$. The inconsistent system of equations represented by the matrix equation*

$$A\mathbf{x} = \mathbf{b}$$

has a least squares approximation determined by the solution of the normal equations,

$$(A^T A)\bar{\mathbf{x}} = A^T \mathbf{b}.$$

The residual vector $\mathbf{r} = \mathbf{b} - A\bar{\mathbf{x}}$ is the vector of minimum magnitude in the set

$$\{\mathbf{b} - A\mathbf{x} : \mathbf{x} \in \mathbb{R}^n\}.$$

8.2 Application of least squares to curves of best fit

The method of least squares applies directly to the originally stated problem of determining a curve of best fit; rather than working an example with a large number of points, let's suppose we have a set of five points through which we desire to find the best-fit parabola².

Example 8.7. Suppose we want to best fit a parabola to the points $(-2, 1)$, $(-1, 2)$, $(0, -2)$, $(1, 1)$, and $(3, 3)$. If such a parabola were to exist, its quadratic equation $y = a_0 + a_1x + a_2x^2$ would need to simultaneously satisfy

$$\begin{cases} a_0 - 2a_1 + 4a_2 = 1 \\ a_0 - a_1 + a_2 = 2 \\ a_0 = -2 \\ a_0 + a_1 + a_2 = 1 \\ a_0 + 3a_1 + 9a_2 = 3 \end{cases} \iff A\mathbf{x} = \begin{bmatrix} 1 & -2 & 4 \\ 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 3 & 9 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ -2 \\ 1 \\ 3 \end{bmatrix} = \mathbf{b}. \quad (8.2)$$

So we obtain an inconsistent system of equations which can be directly translated to the least squares problem, the solution to which gives the coefficients of the parabola of best fit. We obtain

$$A^T A = \begin{bmatrix} 5 & 1 & 15 \\ 1 & 15 & 19 \\ 15 & 19 & 99 \end{bmatrix} \quad A^T \mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 34 \end{bmatrix},$$

so augmenting gives

$$\left[\begin{array}{ccc|c} 5 & 1 & 15 & 5 \\ 1 & 15 & 19 & 6 \\ 15 & 19 & 99 & 34 \end{array} \right] \xrightarrow{rref} \left[\begin{array}{ccc|c} 1 & 0 & 0 & -\frac{67}{679} \\ 0 & 1 & 0 & -\frac{85}{1358} \\ 0 & 0 & 1 & \frac{503}{1358} \end{array} \right].$$

Thus the graph of

$$y = -\frac{67}{679} - \frac{85}{1358}x + \frac{503}{1358}x^2$$

is the best-fit parabola for the five given points. \triangleleft

²The curve represented by a polynomial of degree n can be determined uniquely by a set of exactly $n+1$ points; if additional points are specified, the system (in general) becomes overdetermined and inconsistent.

8.3 Gram-Schmidt orthonormalization

A first method for producing an orthogonal matrix from an arbitrary matrix is the Gram-Schmidt algorithm. In essence, the idea is to treat each column of the initial matrix as a vector, and proceed through in order orthogonalizing each vector to those which have previously been orthonormalized.

Example 8.8. Suppose we begin with two vectors,

$$\begin{aligned}\mathbf{x}_1 &= \langle 1, 1, 1, 1, 1 \rangle \\ \mathbf{x}_2 &= \langle -2, -1, 0, 1, 3 \rangle,\end{aligned}$$

both in \mathbb{R}^5 . Then we can imagine somehow in \mathbb{R}^5 that these two vectors uniquely determine a plane – if we let the “positive horizontal axis” in that plane be determined by the direction of \mathbf{x}_1 and the “positive vertical axis” by \mathbf{x}_2 , we have to determine the projection of \mathbf{x}_2 onto \mathbf{x}_1 . Specifically, we want the “vertical axis” to be in the direction of $\mathbf{x}_2 - \text{proj}_{\mathbf{x}_1} \mathbf{x}_2$, the

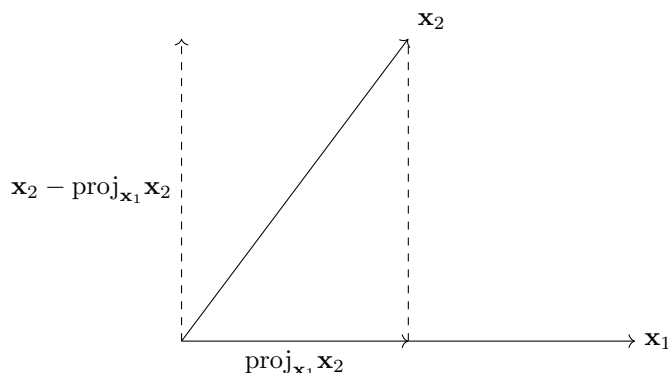


Figure 8.2: Two vectors and the plane they form.

component of \mathbf{x}_2 orthogonal to the projection of \mathbf{x}_2 onto \mathbf{x}_1 , $\text{proj}_{\mathbf{x}_1} \mathbf{x}_2$. For an illustration, see Figure 8.2. \triangleleft

What is very intuitive about the Gram-Schmidt process is how it extends: to add a third vector \mathbf{x}_3 and orthonormalize it, we subtract from \mathbf{x}_3 its projections onto the two previously-determined “axes.” To add a fourth, we subtract its projection onto each of three previously determined axes. The process continues iterating through the n vectors which form the columns of matrix X .

The formula for the projection of a vector \mathbf{b} onto a vector \mathbf{a} follows from the correspondence between the dot product and cosine of the interior angle of two vectors: $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \alpha$, where $0 \leq \alpha \leq \pi$ is the angle between \mathbf{a} and \mathbf{b} . Thus

$$\text{proj}_{\mathbf{a}} \mathbf{b} = (|\mathbf{b}| \cos \alpha) \frac{\mathbf{a}}{|\mathbf{a}|} = \left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}|} \right) \frac{\mathbf{a}}{|\mathbf{a}|}.$$

If it so happens that $|\mathbf{a}| = 1$, then \mathbf{a} is a *unit vector*, and $\text{proj}_{\mathbf{a}} \mathbf{b} = (\mathbf{a} \cdot \mathbf{b})\mathbf{a}$.

Theorem 8.9 (Gram-Schmidt Orthonormalization). *Suppose $S = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ is a set of vectors in \mathbb{R}^m . Suppose*

$$\begin{aligned}\mathbf{w}_1 &= \mathbf{x}_1, \text{ and} \\ \mathbf{q}_1 &= \mathbf{w}_1 / |\mathbf{w}_1|.\end{aligned}$$

Then for each $i = 2, 3, \dots, n$, let

$$\begin{aligned}\mathbf{w}_i &= \mathbf{x}_i - \sum_{j=1}^{i-1} (\mathbf{q}_j \cdot \mathbf{x}_i) \mathbf{q}_j, \text{ and} \\ \mathbf{q}_i &= \mathbf{w}_i / |\mathbf{w}_i|.\end{aligned}$$

Then $Q = \{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n\}$ is an orthonormal basis for $\text{span}(S)$.

Proof. Suppose $1 \leq k < \ell \leq n$. Then

$$\begin{aligned}\mathbf{q}_k \cdot \mathbf{w}_\ell &= \mathbf{q}_k \cdot \mathbf{x}_\ell - \sum_{j=1}^{\ell-1} (\mathbf{q}_j \cdot \mathbf{x}_\ell) \mathbf{q}_j \\ &= \mathbf{q}_k \cdot \mathbf{x}_\ell - \sum_{j=1}^{\ell-1} (\mathbf{q}_j \cdot \mathbf{x}_\ell) (\mathbf{q}_k \cdot \mathbf{q}_j)\end{aligned}$$

Since

$$\mathbf{q}_k \cdot \mathbf{q}_j = \begin{cases} 0 & k \neq j \\ 1 & k = j, \end{cases}$$

we have $\mathbf{q}_k \cdot \mathbf{w}_\ell = \mathbf{q}_k \cdot \mathbf{x}_\ell - \mathbf{q}_k \cdot \mathbf{x}_\ell = 0$. Thus $\mathbf{q}_k \cdot \mathbf{q}_\ell = \frac{1}{|\mathbf{w}_\ell|} (\mathbf{q}_k \cdot \mathbf{w}_\ell) = 0$ and \mathbf{q}_k is orthogonal to \mathbf{q}_ℓ whenever $k < \ell$. Moreover,

$$|\mathbf{q}_k| = \left| \frac{\mathbf{w}_k}{|\mathbf{w}_k|} \right| = 1,$$

so Q is a set of mutually orthogonal unit vectors. Since $\mathbf{w}_\ell = |\mathbf{w}_\ell| \mathbf{q}_\ell$, each vector \mathbf{x}_ℓ can be written as a linear combination from $\{\mathbf{q}_k : k \leq \ell\}$; hence any linear combination of vectors in S can be written as a linear combination of vectors in Q , and vice versa. Thus $\text{span } S = \text{span } Q$ and the result holds. \square

The Gram-Schmidt algorithm actually follows directly from the statement of the theorem and the described process, with one minor modification to improve the numerical stability of the algorithm.

Algorithm 8.10 (Modified Gram-Schmidt Orthonormalization). Let

$$S = \{\mathbf{x}_j : 1 \leq j \leq n\}$$

be a set of vectors in \mathbb{R}^m .

1. For each j in $\{1, 2, \dots, n\}$, set $\mathbf{w}_j = \mathbf{x}_j$. Some of these will be updated later.

2. For j in $\{1, 2, \dots, n\}$, do the following.
 - a. For each $i < j$, set $\mathbf{w}_j = \mathbf{w}_j - (\mathbf{q}_i \cdot \mathbf{w}_j)\mathbf{q}_i$.
 - b. Let $\mathbf{q}_j = \frac{1}{|\mathbf{w}_j|}\mathbf{w}_j$.

The set $Q = \{\mathbf{q}_j : 1 \leq j \leq n\}$ is an orthonormal basis for $\text{span}(S)$.

8.4 QR factorization

Consider a $m \times n$ matrix A ; the columns of A can be considered as n vectors in \mathbb{R}^m . If those vectors are linearly independent, they span an n -dimensional subspace of \mathbb{R}^m . The process of Gram-Schmidt orthonormalization performed upon that set of vectors will result in an orthonormal basis for that subspace. Collecting these vectors as the column vectors of a matrix Q gives us the first half of the QR factorization of A . The matrix R is produced as an upper-triangular matrix recording the magnitudes of the projections removed in each iteration of step 2a above, along with the scaling factors of step 2b. That is, $R = [r_{i,j}]$ is a $n \times n$ matrix with

$$r_{i,j} = \begin{cases} \mathbf{q}_i \cdot \mathbf{w}_j, & i < j \\ |\mathbf{w}_j|, & i = j \\ 0, & i > j. \end{cases} \quad (8.3)$$

This means that the order in which elements of R are recorded during the algorithm will be sort of “transpose” to the normal order: $r_{1,1}, r_{1,2}, r_{2,2}, r_{1,3}, r_{2,3}, r_{3,3}$, and so on.

The matrices Q and R produced as just described are the *reduced QR factorization* of A . If it exists, it is customary to add $m - n$ extra columns to A ; these can be any column vectors linearly independent from the original columns of A , so it is often the case that the columns added correspond to vectors $\langle 1, 0, 0, \dots \rangle$, $\langle 0, 1, 0, \dots \rangle$, $\langle 0, 0, 1, \dots \rangle$, and so on. The orthonormal basis will then be a full basis for \mathbb{R}^m containing m vectors; in order that $A = QR$, it must be that R is a $m \times n$ matrix. In fact this is the case, with $R = [r_{i,j}]$ the $m \times n$ matrix with the same formula from Equation 8.3 determining its elements. This different factorization is called the *full QR factorization* of A .

Definition 8.11. A square matrix Q is *orthogonal* if and only if $Q^T = Q^{-1}$.

Theorem 8.12. Suppose $QR = A$ is the full QR factorization of A . Then Q is an orthogonal matrix.

Proof. Consider $Q^T Q = [m_{i,j}]$. Then

$$m_{i,j} = \mathbf{q}_i \cdot \mathbf{q}_j = \begin{cases} 0 & i \neq j \\ 1 & i = j, \end{cases}$$

and the desired result holds. □

Utility of QR factorization

While we can use the QR factorization to solve a system of equations, it is approximately three times more complicated than LU decomposition. However, QR decomposition can avoid a problem called *ill-conditioning* in solving the least squares problem.

Algorithm 8.13 (Least squares by QR factorization). Given the $m \times n$ inconsistent system of equations represented by $A\mathbf{x} = \mathbf{b}$, let $QR = A$ be the full QR factorization of A and let

$$\hat{R} = \text{upper } n \times n \text{ submatrix of } R$$

$$\hat{\mathbf{d}} = \text{upper } n \text{ entries of } \mathbf{d} = Q^T \mathbf{b}$$

Then the solution to $\hat{R}\bar{\mathbf{x}} = \hat{\mathbf{d}}$ is the least squares solution $\bar{\mathbf{x}}$.

8.5 QR factorization by Householder reflectors

Another algorithm for determining the QR factorization of a matrix which requires fewer operations and is less susceptible to rounding error is the method of Householder reflectors. The method depends upon the following theorem.

Remark. Throughout this section, it will be important to remember that if $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$, then $\mathbf{x}_1^T \mathbf{x}_2 = \mathbf{x}_1 \cdot \mathbf{x}_2$. The transpose notation makes certain calculations and formulas nicer, or easier to prove.

Theorem 8.14. Suppose $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$ have $|\mathbf{x}| = |\mathbf{w}|$. Then $\mathbf{w} - \mathbf{x}$ and $\mathbf{w} + \mathbf{x}$ are orthogonal.

Proof. The proof is made visually in Figure 8.3, but is nonetheless simple to show algebraically:

$$(\mathbf{w} - \mathbf{x})^T (\mathbf{w} + \mathbf{x}) = \mathbf{w}^T \mathbf{w} - \mathbf{x}^T \mathbf{w} + \mathbf{w}^T \mathbf{x} - \mathbf{x}^T \mathbf{x} = |\mathbf{w}|^2 - |\mathbf{x}|^2 = 0.$$

Geometrically, the result follows as the vectors \mathbf{x} and \mathbf{w} form two legs of an isosceles triangle, $\mathbf{w} - \mathbf{x}$ the base, and $\mathbf{w} + \mathbf{x}$ the angle bisector of the angle subtended by the legs. \square

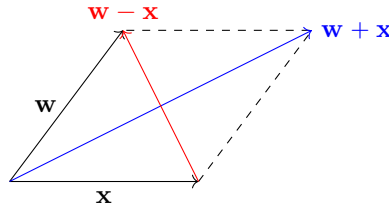


Figure 8.3: Orthogonality of $\mathbf{w} - \mathbf{x}$ and $\mathbf{w} + \mathbf{x}$ in \mathbb{R}^n .

Give two such vectors $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$, define $\mathbf{v} = \mathbf{w} - \mathbf{x}$, and consider the matrix

$$P = \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}.$$

While it may be confusing to think that this is a matrix, recall: since \mathbf{v} is an $n \times 1$ column matrix, $\mathbf{v}\mathbf{v}^T$ is an $n \times n$ symmetric matrix. In fact,

$$P^2 = \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}} \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}} = \frac{\mathbf{v}(\mathbf{v}^T\mathbf{v})\mathbf{v}^T}{(\mathbf{v}^T\mathbf{v})^2} = \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}} = P, \text{ and } P\mathbf{v} = \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}\mathbf{v} = \frac{\mathbf{v}(\mathbf{v}^T\mathbf{v})}{\mathbf{v}^T\mathbf{v}} = \mathbf{v}.$$

A matrix satisfying $P^2 = P$ is called a *projection matrix*. The purpose of using P is that $\mathbf{x} - P\mathbf{x}$ is the projection of \mathbf{x} onto $\mathbf{y} + \mathbf{x}$, and therefore $\mathbf{x} - 2P\mathbf{x} = \mathbf{w}$!

Theorem 8.15. Let $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$ with $|\mathbf{x}| = |\mathbf{w}|$, and suppose $\mathbf{v} = \mathbf{w} - \mathbf{x}$. Then the matrix

$$H = I - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}$$

is a Householder reflector, and is a symmetric orthogonal matrix with $H\mathbf{x} = \mathbf{w}$.

For brevity, I'll refer to the QR factorization by Householder reflectors as HHQR. The process of HHQR for a matrix A iterates through the columns of A just like Gram-Schmidt, but with far less numerical instability. We'll explain the process without use of an example, as the process becomes extremely unwieldy in exact arithmetic.

Suppose we begin with a $m \times n$ matrix³ $A = [a_{i,j}]$, where $m > n$. We will choose as our first vector $\mathbf{x}_1 = \langle a_{1,1}, a_{2,1}, \dots, a_{m,1} \rangle$, the first column of A . Since we desire to reflect it onto a coordinate axis vector of the same magnitude, we can choose $\mathbf{w}_1 = \langle \pm |\mathbf{x}_1|, 0, 0, \dots, 0 \rangle$. Since subtracting nearly identical floating point numbers is problematic and leads to rounding error, we will choose the sign of the first coordinate of \mathbf{w}_1 to be the opposite of the sign of the first coordinate of \mathbf{x}_1 . This allows us to determine the first Householder reflector H_1 such that $H_1\mathbf{x}_1 = \mathbf{w}_1$. Now, putting $R_1 = H_1A$ gives us a matrix whose first column is \mathbf{w}_1 , by construction; notably, this is the first iteration and in the first column R_1 is the start of an upper triangular matrix. Specifically, if we write $R_1 = [r_{i,j}]$, we have $r_{i,1} = 0$ whenever $i > 1$.

In order to continue this process, we will produce from R_1 a new matrix $R_2 = [r_{i,j}]$ which has $r_{i,j} = 0$ whenever $i > j$ and $j \leq 2$. We do so by ignoring the first row and column of R_1 ; doing so gives us R'_1 , an $(m-1) \times (n-1)$ matrix. If we let \mathbf{x}_2 be the first column of R'_1 and $\mathbf{w}_2 = \langle \pm |\mathbf{x}_2|, 0, 0, \dots, 0 \rangle$, we can find the Householder reflector \hat{H}_2 such that $\hat{H}_2\mathbf{x}_2 = \mathbf{w}_2$. Say then that $\hat{H}_2 = [\hat{h}_{i,j}]$. We will create our "real" reflection matrix by inserting \hat{H}_2 into the bottom right corner of an identity matrix. Here is the formula for $h_{i,j}$ along with the more intuitive picture diagram:

$$h_{i,j} = \begin{cases} 1, & i < 2 \text{ or } j < 2, \text{ and } i = j \\ 0, & i < 2 \text{ or } j < 2, \text{ and } i \neq j, \\ \hat{h}_{i,j}, & i, j \geq 2 \end{cases} \quad \text{or} \quad H_2 = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & \hat{H}_2 & \\ 0 & & & \end{array} \right].$$

³The discussion of the method contained here originally appears in Timothy Sauer's excellent *Numerical Analysis*, 2nd edition.

Then $R_2 = H_2 R_1 = H_2 H_1 A$.

We continue this process in the way established: at each step, we reflect the first column of a submatrix onto a coordinate-axis vector of the same length, flipping signs of the first coordinate, and after processing n columns in this way we have $R = H_n H_{n-1} \cdots H_2 H_1 A$, which is an upper triangular matrix. Moreover, the matrices H_j are all orthogonal matrices, so $H_j^{-1} = H_j$. Hence we obtain

$$QR = (H_1 H_2 \cdots H_{n-1} H_n) R = A.$$

Algorithm 8.16 (HHQR). Suppose A is a $m \times n$ matrix.

1. Let Q be the $m \times m$ identity matrix.
2. Let R be a floating point copy of A .
3. In the j^{th} column, for $j = 1, 2, \dots, n$:
 - a. Let $\mathbf{x} = \langle r_{j,j}, r_{j+1,j}, \dots, r_{m,j} \rangle$
 - b. Let $\mathbf{w} = \langle \pm |\mathbf{x}|, 0, 0, \dots, 0 \rangle$ with the first coordinate having the opposite sign as in \mathbf{x} .
 - c. Let $\mathbf{v} = \mathbf{w} - \mathbf{x}$ and $\hat{H} = I - 2 \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T \mathbf{v}}$.
 - d. Let H be the $m \times m$ identity matrix with \hat{H} replacing its bottom right corner.
 - e. Set $Q = QH$
 - f. Set $R = HR$

When the algorithm terminates, $QR = A$.

Here's Python code which implements HHQR when A is specified as a list of lists rather than a matrix. Notice how local functions `sign`, `householder`, and `times` are defined within the `hhqr` function.

```
from math import sqrt

def hhqr(A):
    sign = lambda x: 1 if x>0 else (-1 if x<0 else 0)

    def householder(xvec,wvec):
        vvec = [wvec[i]-x for i,x in enumerate(xvec)]
        d = sum([v**2 for v in vvec])
        H = [[(1 if i==j else 0) - 2*vvec[i]*vvec[j]/d for
              j in range(len(vvec))] for i in range(len(vvec))]
        return H

    def times(A,B):
        return [[sum([A[i][k]*B[k][j] for k in range(len(A[0]))]) for
                j in range(len(B[0]))] for i in range(len(A))]

    m = len(A)
    n = len(A[0])
```

```

Q = [[(1 if i==j else 0) for j in range(m)] for i in range(m)]
R = [[float(x) for x in row] for row in A]
for j in range(n):
    x = [R[i][j] for i in range(j,m)]
    w = [-sign(x[0])*sqrt(sum([xi**2 for xi in x]))+(m-j-1)*[0]
    hh = householder(x,w)
    H = [[(1 if k==l else 0) if (k<j or l<j) else hh[k-j][l-j] for
           l in range(m)] for k in range(m)]
    Q = times(Q,H)
    R = times(H,R)
#####
## Everything after this is just to print the verification output nicely.
print('Verification: Q*R-A\n')
mat = times(Q,R)
mat = [[mat[i][j] - A[i][j] for j in range(n)] for i in range(m)]
clens = [max([len(str(mat[i][j])) for i in range(m)]) for j in range(n)]
out = ''
for i in range(m):
    out += '[ '
    for j in range(n):
        l = len(str(mat[i][j]))
        out += (clens[j]-l)*' '+str(mat[i][j])+', '
    out = out[:-2] + ' ]\n'
print(out)
#####
## Except this, of course.
return Q,R

```


Part III

Introduction to cryptography

Lesson 9

Permutations

Goals

1. Define permutations
2. Permutation operations
3. Implementation of a permutation class

9.1 What is a permutation?

Definition 9.1. Suppose A is a set. Then a function σ with domain A and codomain A is a *permutation of A* if and only if σ satisfies the following two criteria:

- i. (Surjective/Onto) For each element $a \in A$, there is an element $a' \in A$ such that $\sigma(a') = a$.
- ii. (Injective/One-to-one) For any elements $a, b \in A$, if $f(a) = f(b)$ then $a = b$.

Functions which are both one-to-one and onto are also called *bijections*, so an easier definition of a permutation σ of A is that σ is a bijection from A to itself. It is important to note that the definition of a permutation of A does not require that the set A be finite. The study of permutations of infinite sets is beyond the purpose to which we will put our permutations, so we will henceforth make the following assumption: **all sets which we permute will be finite sets.**

Definition 9.2. We make the following notational definitions for the sake of convenience.

- \mathcal{E} is the set of letters in the English alphabet.
- $[n] = \{1, 2, 3, \dots, n\}$ whenever $n \in \mathbb{N}$.

For a finite set A which is not a subset of $[n]$ for any n , we denote the set of all permutations of A by S_A . The set of all permutations of $[n]$ is denoted S_n .

Representing a permutation

There are several distinct ways of representing permutations which are useful in different situations. For non-numeric sets, one of the best is called *two-line notation*.

Example 9.3. Suppose $\sigma_1 : [15] \rightarrow [15]$ is defined by $\sigma_1(i) = 16 - i$. Then the *two line notation* for σ_1 is

$$\sigma_1 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ \hline \end{array}$$

This permutation is one which the “best” representation is its functional one, $\sigma_1(i) = 16 - i$. However, this is not always the case. Consider the following permutation σ_2 of $[15]$ given by

$$\sigma_2 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 4 & 3 & 2 & 7 & 6 & 5 & 10 & 9 & 8 & 13 & 12 & 11 & 1 & 15 & 14 \\ \hline \end{array}$$

Worse still is this permutation σ_3 of \mathcal{E} :

$$\sigma_3 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline a & b & c & d & e & f & g & h & i & j & k & l & m & n & o & p & q & r & s & t & u & v & w & x & y & z \\ \hline d & k & b & c & j & s & m & n & f & o & a & t & w & v & z & p & i & x & u & r & h & q & y & l & g & e \\ \hline \end{array}$$

Two line notation is especially good when there is no natural order for the symbols in the set being permuted. For example, if $A = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit, \nabla, \surd, \infty\}$, then the following is a valid permutation of A :

$$\sigma_4 = \begin{array}{|c|c|c|c|c|c|c|} \hline \clubsuit & \diamondsuit & \heartsuit & \spadesuit & \nabla & \surd & \infty \\ \hline \diamondsuit & \heartsuit & \spadesuit & \clubsuit & \surd & \infty & \nabla \\ \hline \end{array}$$

◁

For some sets, using both lines of the two-line notation is unnecessary. For instance, every permutation in S_n has the same first row in two-line notation! For sets such as these, where there is a commonly understood order and first element, we can suppress the first line of two-line notation, and obtain *one-line notation*.

Example 9.4. We can write the permutations σ_1 and σ_2 above easily in one-line notation¹:

$$\begin{aligned} \sigma_1 &= 15 \ 14 \ 13 \ 12 \ 11 \ 10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1, \\ \sigma_2 &= 4 \ 3 \ 2 \ 7 \ 6 \ 5 \ 10 \ 9 \ 8 \ 13 \ 12 \ 11 \ 1 \ 15 \ 14. \end{aligned}$$

Since σ_3 is a permutation of \mathcal{E} , its one-line notation is a collection of English letters – an unpronounceable word:

$$\sigma_3 = \text{dkbcjsmfnfoatwvzpixurhqylge}.$$

Since the set A defined in Example 9.3 has no clear order, we cannot use one-line notation to represent σ_4 . ▷

¹There is considerable disagreement in the literature about whether the symbols in one-line notation should be separated by spaces, commas, or not at all. Anecdotally, Dr. Kassie Archer of UT Tyler, who does research in combinatorial permutation theory, expressed that her personal preference is to use no separation if there are enough distinct elements in the set, and spaces otherwise. Our notation will follow her example.

Definition 9.5. Suppose A is a set, and let $e_A : A \rightarrow A$ be the function such that $e_A(a) = a$ for each $a \in A$. Then e_A is the *identity permutation on A* . When the set A being permuted is clear by context, the identity permutation will simply be denoted e .

9.2 Permutation groups

Since permutations of a set A are bijections from A to A , they can be composed.

Theorem 9.6. Suppose $\sigma, \tau \in S_A$ for some set A . Then the composition of τ after σ is a permutation $\tau \circ \sigma : A \rightarrow A$ defined by

$$(\tau \circ \sigma)(a) = \tau(\sigma(a))$$

for each $a \in A$. Moreover, the composition symbol \circ will be suppressed, and we will write $\tau\sigma$ as a product rather than a composition².

Two-line notation makes calculating the composition of permutations uncomplicated, if the permutations are “stacked” in the correct order.

Example 9.7. Consider σ_1 and σ_2 from Example 9.3. To determine the product $\sigma_2\sigma_1$, we write σ_1 first and then σ_2 below it, in two-line notation. The output of σ_1 then becomes the input of σ_2 , as demonstrated below by an arrow.

$$\sigma_1 = \begin{array}{c|cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{array}$$

↓

$$\sigma_2 = \begin{array}{c|cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 4 & 3 & 2 & 7 & 6 & 5 & 10 & 9 & 8 & 13 & 12 & 11 & 1 & 15 & 14 \end{array}$$

If each possible such arrow is followed, the result can be tabulated in the two-line notation for $\sigma_2\sigma_1$,

$$\sigma_2\sigma_1 = \begin{array}{c|cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 14 & 15 & 1 & 11 & 12 & 13 & 8 & 9 & 10 & 5 & 6 & 7 & 2 & 3 & 4 \end{array}.$$

As a memory aid the product $\sigma_2\sigma_1$ should be read as “ σ_2 of σ_1 ,” rather than “ σ_2 times σ_1 ,” much as with fractions: $\frac{3}{2}(15)$ can be read as “three halves of fifteen.” \triangleleft

Next we know from elementary calculus that a function which is one-to-one must have an inverse.

Theorem 9.8. Suppose $\sigma \in S_A$. Then there is a unique inverse for σ , denoted σ^{-1} , such that $\sigma^{-1}\sigma = e_A = \sigma\sigma^{-1}$.

Example 9.9. We need to note that composition is **not commutative**. Generally speaking, $\sigma\tau \neq \tau\sigma$. To demonstrate, observe

$$\sigma_1\sigma_2 = \begin{array}{c|cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 12 & 13 & 14 & 9 & 10 & 11 & 6 & 7 & 8 & 3 & 4 & 5 & 15 & 1 & 2 \end{array}.$$

²This is familiar: think of $f(x) = \arctan \cos(x)$.

Clearly $\sigma_1\sigma_2 \neq \sigma_2\sigma_1$, since two permutations can only be equal if they have the same two-line notation. \triangleleft

Composition of permutations satisfies another desirable property: it is associative. That is, $\sigma(\tau\phi) = (\sigma\tau)\phi$.

Definition 9.10. A set G along with an associative operation $*$ which is closed under inverses and has an identity element is a *group*. The set S_A with composition is thus called the *symmetric group over A* .

Remark. In fact, the study of group theory (which begins for modern students in Math 3336) was inspired by the work of Lagrange on generalizations of permutation groups. An important result in group theory, especially for those working with groups in an applied setting, is *Cayley's Theorem*: every abstract group is isomorphic to a subgroup of a symmetric group.³

9.3 Cycles and disjoint cycle decomposition

There is another method of writing permutations which is preferred for group theory as it makes prominent special features of permutations.

Definition 9.11. Suppose $a \in A$ and $\sigma \in S_A$. Then σ *fixes* a , or equivalently a is a *fixed point* of σ , if and only if $\sigma(a) = a$. Elements of A which are not fixed by σ are *moved* by σ .

Definition 9.12. Let i_1, i_2, \dots, i_r be distinct integers in $[n]$, where $n \geq r$. Suppose $\alpha \in S_n$ fixes all integers in $[n] \setminus \{i_1, i_2, \dots, i_r\}$ and also that

$$\alpha(i_1) = i_2, \quad \alpha(i_2) = i_3, \quad \dots \quad \alpha(i_{r-1}) = i_r, \quad \alpha(i_r) = i_1.$$

Then α is an *r -cycle*, or equivalently a *cycle of length r* , and is denoted in *cycle notation* by

$$\alpha = (i_1 \ i_2 \ \dots \ i_r).$$

An interesting result of the definition of cycles is that every 1-cycle is equivalent to the identity: if $a \in [n]$ and $\alpha = (a)$, then $\alpha(a) = a$ and for all $b \in [n] \setminus \{a\}$, b is fixed by α , so $\alpha(b) = b$. Thus $(a) = e$ for any $a \in [n]$.

Definition 9.13. Two cycles α and β are *disjoint* if and only if every element moved by α is fixed by β and every element moved by β is fixed by α .

Lemma 9.14. If α and β are disjoint cycles in S_n , then $\alpha\beta = \beta\alpha$.

Theorem 9.15. Every permutation is either a cycle or a product of disjoint cycles.

Moreover, this decomposition into disjoint cycles can be written uniquely up to the order in which the cycles appear!

³Don't worry about "isomorphic." If two groups are said to be isomorphic, you can just think that each group is a "creative relabelling" of the other.

Example 9.16. Consider again σ_2 from Example 9.3,

$$\sigma_2 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 4 & 3 & 2 & 7 & 6 & 5 & 10 & 9 & 8 & 13 & 12 & 11 & 1 & 15 & 14 \\ \hline \end{array}.$$

We can write the cycle decomposition of σ_2 by starting with 1: the first cycle of the decomposition will be $(1 \ \sigma_2(1) \ \sigma_2^2(1) \ \cdots \ \sigma_2^s(1))$ for some $s \leq n$. In fact,

$$\sigma_2 = (1 \ 4 \ 7 \ 10 \ 13)(2 \ 3)(5 \ 6)(8 \ 9)(11 \ 12)(14 \ 15).$$

While cycle notation is useful for many purposes, some find it difficult to multiply permutations written in their cycle decompositions; the rule to follow is that the “motion” of symbols must be evaluated from right to left. That is, to discover the value of $\sigma_2^2(1)$, we write

$$\sigma_2^2 = (1 \ 4 \ 7 \ 10 \ 13)(2 \ 3)(5 \ 6)(8 \ 9)(11 \ 12)(14 \ 15)(1 \ 4 \ 7 \ 10 \ 13)(2 \ 3)(5 \ 6)(8 \ 9)(11 \ 12)(14 \ 15)$$

and proceeding from right to left, we find that $1 \mapsto 4 \mapsto 7$, $2 \mapsto 3 \mapsto 2$, and so on. Hence

$$\sigma_2^2 = (1 \ 7 \ 13 \ 4 \ 10)(2)(3)(5)(6)(8)(9)(11)(12)(14)(15) = (1 \ 7 \ 13 \ 4 \ 10),$$

since the 1-cycles act as e . \triangleleft

Definition 9.17. Suppose $\sigma \in S_A$ is a permutation. Then the *order* of σ , denoted $|\sigma|$, is the smallest positive integer n such that $\sigma^n = e$.

Lemma 9.18. Suppose $\alpha \in S_A$ is an r -cycle. Then $|\alpha| = r$.

Theorem 9.19. If $|S_A|$ is the number of permutations⁴ of the set A and $\sigma \in S_A$, then there is an integer k such that $k|\sigma| = |S_A|$.

We are now equipped to see the real beauty of the cycle decomposition of a permutation.

Theorem 9.20. Suppose $\sigma \in S_A$ and $\sigma = \alpha_1 \alpha_2 \cdots \alpha_k$ is its cycle decomposition. Then

$$|\sigma| = \text{lcm} \{ |\alpha_i| : i \in [k] \}.$$

Example 9.21. Another beauty of cycle notation is that we see the behavior of certain powers of a permutation quite clearly. For example, if we let

$$\phi = (1 \ 3 \ 7 \ 5)(2 \ 4 \ 6 \ 8 \ 9),$$

we can write $\alpha = (1 \ 3 \ 7 \ 5)$ and $\beta = (2 \ 4 \ 6 \ 8 \ 9)$. Then $|\alpha| = 4$ and $|\beta| = 5$, and

$$\phi^5 = \alpha^5 \beta^5 = \alpha^4 \alpha e = \alpha$$

and

$$\phi^4 = \alpha^4 \beta^4 = e \beta^4 = \beta^{-1}.$$

\triangleleft

⁴If there are n distinct elements in A , then $|S_A| = n!$.

9.4 Implementing permutations in Python

Dictionaries

We've talked about `lists` and `tuples`, both compound data types which are indexed by nonnegative integers. When working with S_A for arbitrary sets A , it becomes necessary to index by elements of A . In Python the mechanism to do this is called a *dictionary*, with data type `dict`. Dictionaries can be defined explicitly as follows:

```
>>> d = {'bob': (3,2,1), 2:'larry', (1,5):['this is data', 3]}
>>> d['bob']
(3, 2, 1)
>>> d[2]
'larry'
>>> d[(1,5)]
['this is data', 3]
>>> d.keys()
dict_keys(['bob', 2, (1, 5)])
>>> d.values()
dict_values([(3, 2, 1), 'larry', ['this is data', 3]])
>>> d.items()
dict_items([('bob', (3, 2, 1)), (2, 'larry'), ((1, 5), ['this is data', 3])])
```

So we see that the valid indices of a `dict` are called its **keys**, and the values indexed by those keys are called **values**. The `tuples`

`('bob', (3, 2, 1)), (2, 'larry'), ((1, 5), ['this is data', 3])`

given in the final evaluation give another way to define a `dict`. Interestingly, the key-value pairs in a dictionary are stored in a **set**, a Python data type which behaves as a set does in abstract mathematics.

```
>>> d = dict([(2*x, list(range(-x,x+1))) for x in range(10)])
>>> for x,y in d.items():
    print(x,": ",y)

0 :  [0]
16 :  [-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
2 :  [-1, 0, 1]
4 :  [-2, -1, 0, 1, 2]
6 :  [-3, -2, -1, 0, 1, 2, 3]
8 :  [-4, -3, -2, -1, 0, 1, 2, 3, 4]
10 : [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
12 : [-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6]
18 : [-9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
14 : [-7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
```

In this example we have a dictionary indexed by integers whose values are lists, but we don't need to have all the indices! Hence this is a good way to store “sparse” arrays, where only some of the indices are necessary. A more important feature of dictionaries is that the values can be accessed by indexing in the normal `__getitem__` kind of way:

```
>>> d[4]
[-2, -1, 0, 1, 2]
```

There are lots of different ways to construct a dictionary, but one of the easiest is to use Python's `zip` function. The input to `zip` is any two iterables: the output is an iterable `zip` object. This is like a `range` object, which is really only useful when you iterate over it.

```
>>> a = 'alpha'
>>> b = "beta"
>>> z = zip(a,b)
>>> z
<zip object at 0x10568af48>
>>> for x in z:
    print(x)

('a', 'b')
('l', 'e')
('p', 't')
('h', 'a')
```

Effectively, this provides a tuple of ordered pairs. The zipped object will only have length equal to the length of the shorter inputs, but this is very useful if they are the same length. In fact, we can use `zip` to construct the list of tuples which is an option for instantiating a dictionary.

```
>>> a = 'abcdefghijklmnopqrstuvwxyz'
>>> b = 'thequickbrownfxjmpsvlazydg'
>>> d = dict(zip(a,b))
>>> d
{'s': 's', 'k': 'o', 't': 'v', 'o': 'x', 'v': 'a', 'i': 'b', 'g': 'c', 'x': 'y',
 'w': 'z', 'a': 't', 'z': 'g', 'l': 'w', 'y': 'd', 'u': 'l', 'f': 'i', 'h': 'k',
 'j': 'r', 'd': 'q', 'c': 'e', 'q': 'm', 'b': 'h', 'r': 'p', 'p': 'j', 'e': 'u',
 'm': 'n', 'n': 'f'}
```

This⁵ is particularly useful as we can now use indexing sort of as a function call to our permutation, but we will have to be very careful about punctuation.

```
>>> p = 'this is a plain text.'
>>> for x in p:
    print(d[x],end = ' ')

v k b s Traceback (most recent call last):
  File "<pysHELL#40>", line 2, in <module>
    print(d[x],end = ' ')
KeyError: ' '
```

⁵The string used for `b` in the example code is derived from a short sentence using all lowercase characters in English: “The quick brown fox jumps over the lazy dog.” This is often used to demonstrate the shape of all the miniscule characters in a type face.

Further, we want to think of our permutations as functions, not as indexed arrays, so we'd like to call use `d(x)` instead of `d[x]`. In order to do this we'll actually need to build a permutation class and overload the builtin `__call__` operator, which allows functions to be called as functions.

9.5 A basic permutation class

As with all classes, we first have to define the `__init__` and `__repr__` methods, so that we can instantiate `Perm` objects and represent them in the Python interpreter.

Algorithm 9.22 (Input a permutation in one-line notation). We assume that the input is a string of integers. If any of the integers consist of more than one digit, the integers must be separated by spaces.

1. If the input string contains any spaces, let `oneline` be the list of substrings separated by spaces.
2. Otherwise, let `oneline` be the list of single-character substrings.
3. In either case, reset `oneline` to be the list of integer versions of the substrings.
4. Check that the length of `oneline` is equal to `n`, the greatest integer contained in `oneline`; otherwise, the input is not a well defined one-line permutation and a `ValueError` should be raised.
5. Create an empty dictionary `d`
6. For each pair `i,x` in `enumerate(oneline)`, assign the correct key of `d` to the correct value.
7. Assign the attributes of `self`:
 - a. `self._data` should be set to `d`
 - b. `self._is_int` should be `True`; later we will probably want to permute non-integers and it will be important to check this.
 - c. `self._domain` should be the list `[1, 2, ..., max(d.keys())]`. We'll need this for composition, later.

A good exercise at this point is to think about how you would implement one-line permutations if the input is an alphanumeric string.

Exercise 9.23. *Extend the algorithm above so that the input can be any alphanumeric string, as long as each character in the string only occurs once. The output permutation should take the input string and sort it, use that for the sorted list of keys, and assign to each character in the sorted list the corresponding position in the input list. Further, any alphanumeric characters not included in the list should be included in the permutation as fixed points.*

For instance, `Perm('thequ1i2ck3br4o56w7nfx8j9mpsv0lazydg')` should contain a dictionary as follows:


```
{'0': 't', '1': 'h', '2': 'e', '3': 'q', '4': 'u', '5': '1', '6': 'i', '7': '2',
 '8': 'c', '9': 'k', 'a': '3', 'b': 'b', 'c': 'r', 'd': '4', 'e': 'o', 'f': '5',
 'g': '6', 'h': 'w', 'i': '7', 'j': 'n', 'k': 'f', 'l': 'x', 'm': '8', 'n': 'j',
 'o': '9', 'p': 'm', 'q': 'p', 'r': 's', 's': 'v', 't': '0', 'u': '1', 'v': 'a',
 'w': 'z', 'x': 'y', 'y': 'd', 'z': 'g'}
```

As for `__repr__`, since we are working (at this point) with permutations of integers, we should represent a permutation by its cycle decomposition.

Algorithm 9.24 (Disjoint cycle decomposition). Begin with a permutation σ saved as a dictionary `d`.

1. Let `keys` be a sorted copy of the list of keys of `d`
2. Create an empty list `decomp`
3. While `keys` is not empty:
 - a. Start a new list `c` containing the least element of `keys`
 - b. Add elements to `c` in the order $c_1 = \sigma(c_0)$, $c_2 = \sigma(c_1)$, until you find an element c_r such that $\sigma(c_r) = c_0$. In each addition, remove c_i from `keys`.
 - c. If $r > 0$, add a tuple of (c_0, c_1, \dots, c_r) to `decomp`.
4. Return `decomp` nicely as a string of the cycle decomposition of σ . If `decomp` is empty, the return should be the string `"(1)"`.

In fact, it might be better to implement this algorithm as a `Perm.cycles()` method rather than in `Perm.__repr__`, so that non-integer strings can be automatically printed in either one-line or two-line notation.

The next complicated step in defining permutations is deciding how to implement permutation composition as left-multiplication. To make this less complicated, we'll define overload the builtin `__call__` operator (which lets us use the notation $\sigma(x)$) and define a `domain` method to return the value of `self._domain`.

```
def __call__(self, value):
    try:
        return self._data[value]
    except KeyError:
        return value

def domain(self):
    return self._domain
```

This sets us up for the composition algorithm.

Algorithm 9.25 (Permutation composition). Suppose we have two permutations, σ and τ , represented as `Perm` objects by `s` and `t` respectively.

1. Let `dom = list(set(s.domain()).union(t.domain()))`. This takes the domain to be the union of domains of σ and τ , as a list.
2. Sort the domain `dom`.
3. Let `oneline` be a string of values `s(t(x))` for each `x` in `dom`, each separated by a space. This is why we wanted permutations to be callable.
4. Return `Perm(oneline)`.

This should be implemented as `Perm.__mul__`, and since permutations can only be multiplied by other permutations, we need not worry about implementing `Perm.__rmul__`. To have full functionality for our permutations, the last two steps are related: defining the inverse of a permutation and powers of permutations. Our implementation of permutations as dictionaries makes inverses a snap.

```
def inverse(self):
    value = [(y,x) for x,y in self._data.items()]
    value.sort()
    return Perm(' '.join([str(y) for x,y in value]))
```

As a means to implementing powers, we'll also implement the order function for a permutation. Recall that the order of a permutation is the least common multiple of the lengths of the cycles in its disjoint cycle decomposition.

```
def order(self):
    gcd = lambda a,b: b if a%b==0 else gcd(b,a%b)
    def lcm(a_list):
        l = a_list[0]
        for x in a_list[1:]:
            l = (l*x)//gcd(l,x)
        return l
    c = [len(x) for x in [y.split() for y in self.cycles()[1:-1].split(')(')]]
    return lcm(c)
```

We'll use this because we have a nice fact.

Lemma 9.26. *Suppose $\sigma \in S_n$ with $|\sigma| = k$, and suppose $p \in \mathbb{Z}$. Then $\sigma^p = \sigma^{p \bmod k}$.*

```
def __pow__(self,value):
    if type(value)!=int:
        raise TypeError('Only integer powers of permutations are permitted.')
    p = value % self.order()
    outperm = Perm('1')
    for i in range(p):
        outperm = outperm * self
    return outperm
```

Now we can use `g**5` to represent a permutation g^5 , and while this is not a full-featured class of permutations, it is fully usable for our purposes.

Lesson 10

Substitution Ciphers

Goals

1.

Introduction

A *substitution cipher* in its simplest form is simply a permutation of the letters in an alphabet; more complicated substitution ciphers might map single characters to fixed-size blocks of characters.

10.1 Simple substitution

Also called *monoalphabetic substitution*, these are among the easiest encryptions to break. A simple substitution cipher employs a single permutation of the symbol set to encrypt the plaintext. The *Caesar cipher* is one of the earliest known uses: in an alphabet of size n , the permutation for Julius Caesar's cipher was $\sigma(x) = (x + 3) \bmod n$. For Caesar's military dispatches, the permutation would have been

$$\sigma_1 = \begin{array}{cccccccccccccccccccccccccc} \text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} & \text{G} & \text{H} & \text{I} & \text{J} & \text{K} & \text{L} & \text{M} & \text{N} & \text{O} & \text{P} & \text{Q} & \text{R} & \text{S} & \text{T} & \text{U} & \text{V} & \text{W} & \text{X} & \text{Y} & \text{Z} \\ \hline \text{D} & \text{E} & \text{F} & \text{G} & \text{H} & \text{I} & \text{J} & \text{K} & \text{L} & \text{M} & \text{N} & \text{O} & \text{P} & \text{Q} & \text{R} & \text{S} & \text{T} & \text{U} & \text{V} & \text{W} & \text{X} & \text{Y} & \text{Z} & \text{A} & \text{B} & \text{C} \end{array}.$$

Since $26 \equiv 1 \bmod 3$, this permutation σ_1 would be written as a single 26-cycle in disjoint cycle notation.

Generally speaking, we'll call an encryption of this type either a *shift cipher* or a *rotation cipher*; another frequently used rotation cipher is called "ROT13," and is given by

$$\sigma_2 = \begin{array}{cccccccccccccccccccccccccc} \text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} & \text{G} & \text{H} & \text{I} & \text{J} & \text{K} & \text{L} & \text{M} & \text{N} & \text{O} & \text{P} & \text{Q} & \text{R} & \text{S} & \text{T} & \text{U} & \text{V} & \text{W} & \text{X} & \text{Y} & \text{Z} \\ \hline \text{N} & \text{O} & \text{P} & \text{Q} & \text{R} & \text{S} & \text{T} & \text{U} & \text{V} & \text{W} & \text{X} & \text{Y} & \text{Z} & \text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} & \text{G} & \text{H} & \text{I} & \text{J} & \text{K} & \text{L} & \text{M} \end{array}.$$

Since ROT13 is an involution¹, it is used as the canonical example of weak encryption; it was commonly used online in usenet newsgroups during the early 1980s to obscure text such as "spoilers," punchlines to jokes, or obscene language.

¹A permutation σ is an involution if and only if $\sigma^2 = (1)$.

Polygraphic substitution

Another more secure technique is to permute groups of letters, effectively increasing the size of the alphabet. A digraph encryption would take pairs of adjacent letters (*digraphs*) and permute them with another digraph.

In the Playfair cipher² a 5×5 grid of letters containing a keyword is memorized, along with four rules. The keyword is written into the grid, with duplicate letters removed, and the remaining letters follow. The letters *I* and *J* are written into the same position. For instance, if our key phrase is “TALONS UP,” the grid would be

T	A	L	O	N
S	U	P	B	C
D	E	F	G	H
I	J	K	M	Q
R	V	W	X	Y
Z				

The rules are

1. If the paired letters are the same, insert an *X* between them.
2. If the letters occur in the same row of the table, use the letters to their immediate right, wrapping around to the left side of the table if necessary.
3. If the letters occur in the same column of the table, use the letters immediately below them, wrapping around to the top of the table if necessary.
4. Otherwise, draw a rectangle around the pair of letters, and replace each with the unused corner from the same row.

These rules are only difficult to formulate in writing, but in practice are easy to implement; the inventor claimed he could teach it to 3 out of 4 school boys in 15 minutes. To demonstrate the use of the cipher, we'll use our Playfair grid to encrypt the message, HIT THE BOOKKEEPER. First, we break the plaintext into pairs, inserting *X*'s where necessary:

HITTHEBOOKKEEPER \mapsto HI TX TH EB OX OK KE EP ER

Next we draw the appropriate boxes, as demonstrated in Figure 10.1.

The resulting plaintext is DRLVN DGULY AQWKF UHK, broken into 5-character blocks. To decrypt the message is to use the same Playfair grid and reverse rules 2, 3, and 4. Unfortunately, the use of this digraph substitution remains equivalent to a monoalphabetic substitution cipher, this time on an alphabet of $26 \cdot 25 = 650$ symbols, since no double-letter digraphs are permuted. Since $\log_{10}(650!) \approx 1547.91$, this is a considerably large space of permutations to attack naively.

10.2 Polyalphabetic substitution

Polyalphabetic substitution still uses permutations to function, but *the permutation changes between characters*. This presents sufficient difficulty to decryption that

²Developed in 1854 by Charles Wheatstone but advocated by Lord Playfair.

T A L O N	T A L O N	T A L O N
S U P B C	S U P B C	S U P B C
D E F G H	D E F G H	D E F G H
I J K M Q R	I J K M Q R	I J K M Q R
V W X Y Z	V W X Y Z	V W X Y Z
HI→DR	TX→LV	TH→ND

T A L O N	T A L O N	T A L O N
S U P B C	S U P B C	S U P B C
D E F G H	D E F G H	D E F G H
I J K M Q R	I J K M Q R	I J K M Q R
V W X Y Z	V W X Y Z	V W X Y Z
EB→GU	OX→GU	OK→AQ

T A L O N	T A L O N	T A L O N
S U P B C	S U P B C	S U P B C
D E F G H	D E F G H	D E F G H
I J K M Q R	I J K M Q R	I J K M Q R
V W X Y Z	V W X Y Z	V W X Y Z
KE→WK	EP→FU	ER→HK

HITXTHEBOXOKKEEPER→DRLVNDGULYAQWKFUHK

Figure 10.1: Encoding characters using Playfair's cipher.

The Vigenère Cipher

The basis for the Vigenère cipher is the *tabula recta*, a 26×26 matrix of letters as shown in Figure 10.2. Rather than simply using one shift cipher, this tabula recta incorporates every shift cipher. The process for using it is as follows:

1. Select a keyword or keyphrase.
2. Write the plain text in one line.
3. Write the keyphrase repeatedly beneath the plain text.
4. The letter of the keyphrase determines which row of the tabula recta to look at, and the corresponding letter of the plain text determines which column.

For instance, if our key phrase is

"THIS PARROT HAS CEASED TO BE"

and our plain text is

"THATS ONE SMALL STEP FOR A MAN ONE GIANT LEAP FOR MANKIND",

we would write (blocking the message into 5-character blocks for readability)

T H A T S	O N E S M	A L L S T	E P F O R	A M A N O
T H I S P	A R R O T	H A S C E	A S E D T	O B E T H
N E G I A	N T L E A	P F O R M	A N K I N	D
I S P A R	R O T H A	S C E A S	E D T O B	E

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
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	A
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	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

Figure 10.2: The *tabula recta* used in the Vigenère cipher and other polyalphabetic substitutions.

We then proceed to use the tabula recta: since the first character of the key phrase is a T, we look in the row beginning with T; we then find the column beginning with T, as T is the first character of the message. The letter in the T^{th} row and T^{th} column is M. Proceeding in this manner we obtain the cyphertext:

MOILH OEVG F HLDUX EHJRK ONEGV VWVIR EHELA HHSRE EQDWO H.

The Enigma Cipher

Perhaps the most famous modern polyalphabetic substitution cipher is Enigma, used by Nazi Germany during World War II to encrypt its military transmissions. The Enigma machines were not computers – early versions were machines which used a series of rotors to encode messages between a keyboard and a lampboard, while later machines were modified to print messages directly onto a narrow paper strip.

Enigma machines had a varying number of permutation rotors at different stages of the war, as well as a “plugboard”. Each rotor was a plastic disc with 26 spring-loaded electrical contact pins on one side and 26 corresponding contact plates on the other side. Connecting pins on one side to plates on the other was a wiring which corresponds to a permutation of the 26 signals. Assuming a 3-rotor Enigma, the electrical path of a signal would pass from the key to the plugboard, to the entry wheel, then through the three rotors – right, middle,

then left. Finally, there was an element called a reflector, which acted as an *involution*, and then the path went through the rotors again, but reversed – left, right, middle. The electrical signal then would pass again through the plugboard and to the lamps. Both plugboard and reflector were reconfigurable.

Further, when a key was released, the rotation mechanism would engage. The right rotor would step, and upon reaching a notch in its mechanism would make the middle rotor step. This in turn could turn the left rotor in the same way. Of the eight rotors in use during World War II, rotors VI, VII, and VIII each had two such notches, providing an irregular stepping to break up the odometer-style pattern of rotor stepping. The result of this stepping mathematically is equivalent to shifting *just the bottom row* of the two-line notation for that rotor's permutation by one position to the left. It is this stepping which makes Enigma polyalphabetic – the permutation changes with each letter encrypted.

A further feature of the Enigma machine is that every setting of rotors and plugboard results in a *derangement*. This feature was patented prior to World War II, when Enigma was sold commercially, and greatly aided Polish algebraist Marian Rejewski in his work to break the encryption.

Definition 10.1. A *derangement* is a permutation with no fixed points.

Coupling the use of derangements with the knowledge that the same machine had to encrypt and decrypt the message using the same settings, Rejewski and others were able to devise mathematical theory which allowed the Enigma to be broken. For instance, consider the permutations as follows:

$$\begin{aligned} P &= \text{plugboard} \\ R &= \text{right rotor} \\ M &= \text{middle rotor} \\ L &= \text{left rotor} \\ U &= \text{reflector} \end{aligned}$$

Then we know certain things from permutation theory and the design of the Enigma machine. First, we know that P and U are involutions: $P^2 = U^2 = (1)$. Further, we know that

$$E = PRMLUL^{-1}M^{-1}R^{-1}P^{-1}$$

is a derangement, as is

$$E_{i,j,k} = P(\rho^i R \rho^{-i})(\rho^j M \rho^{-j})(\rho^k L \rho^{-k})U(\rho^k L^{-1} \rho^{-k})(\rho^j M^{-1} \rho^{-j})(\rho^i R^{-1} \rho^{-i})P^{-1},$$

the permutation after stepping each rotor the specified number of times. With three rotors installed out of a set of five, 26 different initial rotor settings for each installed rotor, a configurable reflector, and a plugboard with up to ten pairs of letters transposed, the military version of Enigma could be configured to a staggering number of initial settings. Each of these can be considered the “key phrase.”

The use of the initial rotor setting was only intended to be used for transmission of the individual message key: the correct code book setting for the rotors would be set, then some “randomly selected” three-letter *message key* would be selected. This would be sent through the Enigma twice consecutively. Once this was completed, the rotors were set to

the positions of the message key and encoding of the actual message would commence. The message key could be read off the initial rotor settings in normal left-to-right order.

Example 10.2. The actual rotor wirings – and the permutations they produce – can be found by a quick web search. Even more interesting is that key pages from the Luftwaffe survived the war and can be found! The key settings for day 30 of Luftwaffe Machine Key no. 649, reproduced from Wikipedia, specifies rotors I, V, and III, with ring settings 14, 09 24. The reflector setting is KM AX PZ GO DI CN BR PV LT EQ HS UW, and the plugboard is to be set to SZ GT DV KU FO MY EW JN IX LQ. Further, we'll use the first of the ground settings, **wny**. Suppose I choose my message key to be **qzv**. I begin by encoding **qzv** twice under ground setting **wny**, given the correct initialization.

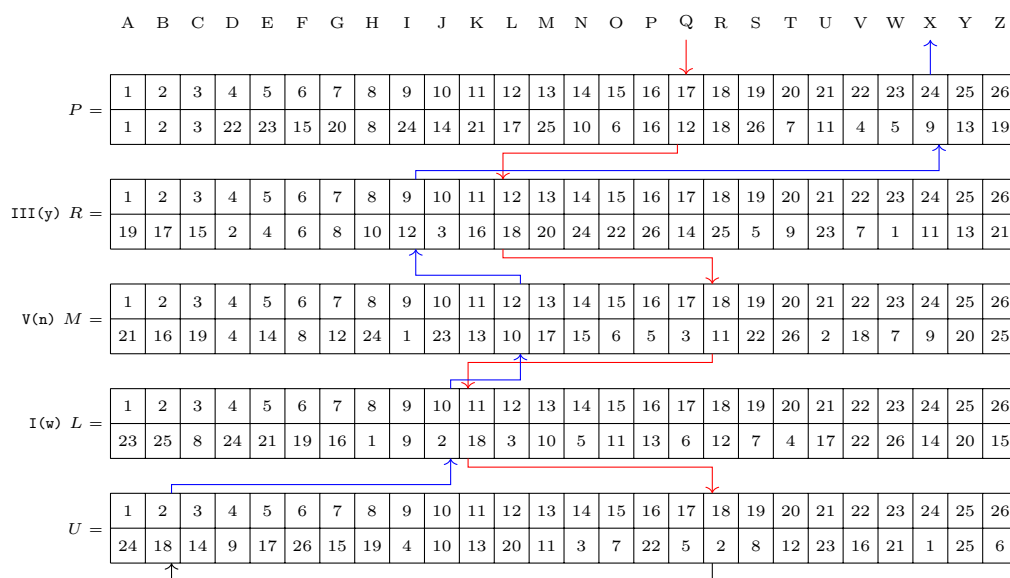


Figure 10.3: The beginning of sending a message indicator using Enigma maps Q to P.

The electrical pathway traced through the plugboard, rotors, and reflector is demonstrated in Figure 10.3. After Q is pressed, X lights up. When Q is released, we see that the right-most rotor is not in position to step its neighbor, so only that rotor will step. The next pathway is demonstrated in Figure 10.4.

Further diagrams can be drawn, demonstrating the turning rotors as the message indicator QZVQZV is encrypted to XNCYFT. The rotors are then manually set to positions QZV and the secret message begins being encrypted. If the full message you intercept is

XNCYF TSIXR VIVGQ MXVFO AHOLR YLZUN EEHDT GOAXT GRYVO
ALGPV OCBK HTDVB VLFKD QPPKV UAYRZ ANYGY IHHFE GFJM

can you decipher it, knowing my initial settings of the Enigma?

◁

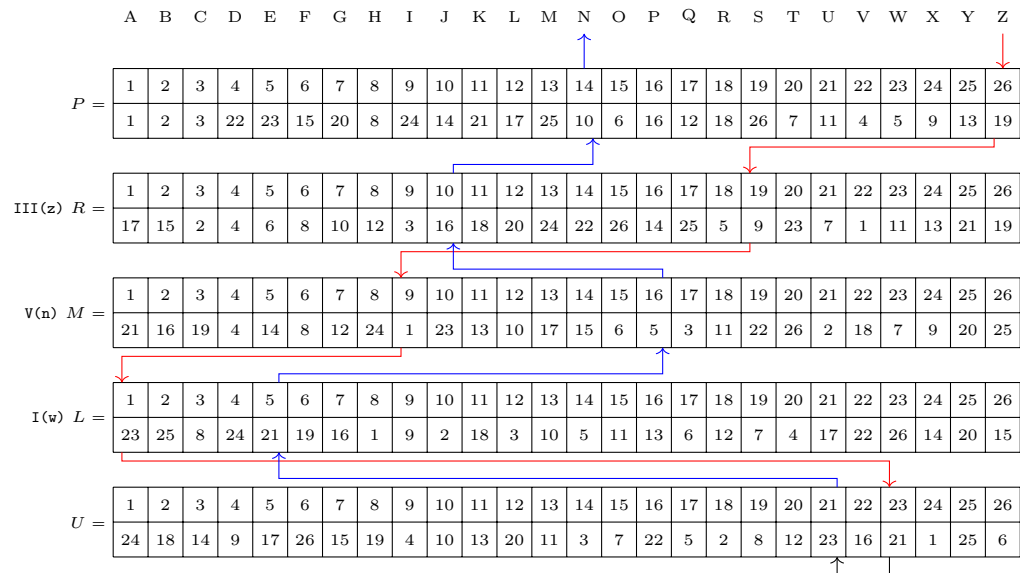


Figure 10.4: The second character encoding of the same message indicator maps Z to N.

Lesson 11

Public Key Encryption

Goals

1. Explain the difference between public key and “shared secret” cryptography.
2. Develop kid-RSA and demonstrate its strength and weakness.
3. Develop RSA encryption.

11.1 Shared secret vs. public key encryption

Throughout Lesson 10, we talked about “shared key” cryptography: if Alice is sending Bob an encrypted message, both Alice and Bob must have the whole key and understand the method. With the advent of digital computers it became necessary to build a stronger type of encryption, where the encryption and decryption keys are distinct. While first theoretically proposed by Diffie and Hellman in 1976, public key encryption saw its first practical application in the algorithm created¹ by Ron Rivest, Adi Shamir, and Leonard Adelman, which now bears their initials: the RSA encryption algorithm.

The difficulty inherent in creating the RSA algorithm involved finding a “one-way function,” a function f which has an inverse $f^{-1} \neq f$ which is computationally difficult to find. The function used in RSA involves modular exponentiation.

Conceptually, shared key cryptography is like the type of padlock which requires the key either to lock or unlock. Anyone with the key can encrypt, and they use the same key to decrypt. In RSA, the encryption key and the decryption key are mathematically related, but not equal. Hence the encryption key can be posted publically – this is why some emails will contain a line in the signature which says “PGP Key.” Pretty Good Privacy, or PGP, is a cryptosystem that uses RSA as one of its steps. The decryption key in public key cryptography is called the *private key* and is never shared.

We’ll discuss an introductory version of RSA (sometimes called Kid RSA) before moving on to the full RSA algorithm. The difference between the two involves a different use of modular arithmetic in the key generation phase.

¹Clifford Cocks of the UK GCHQ described a system equivalent to RSA in 1973, but it was not known until 1997 due to its classification as top-secret.

11.2 Kid RSA

As an introduction to the RSA algorithm, modular multiplication can be used to greatly simplify the calculation process; the trade-off is that the algorithm becomes susceptible to a very well-known mathematical attack. The only *shared secret* in using Kid RSA is the method of converting strings into integers.

Key Generation

The first step of a public key cryptography algorithm is the generation of the public and private keys.

Algorithm 11.1 (Kid RSA Key Generation). Let $a, b, a', b' \in \mathbb{Z}$ be arbitrary positive integers. Define

$$\begin{aligned} M &= ab - 1 \\ e &= a'M + a \\ d &= b'M + b \\ n &= (ed - 1)/M = a'b'M + ab' + a'b + 1 \end{aligned}$$

The public key is the ordered pair (n, e) ; the private key is the ordered pair (n, d) .

Having a public key generated and shared, a user can now wait for an encrypted message to arrive.

Example 11.2. Suppose we choose

$$a = 5 \qquad b = 25 \qquad a' = 7 \qquad b' = -49.$$

Following Algorithm 11.1, we obtain

$$\begin{aligned} M &= 5(25) - 1 = 124 \\ e &= 7(124) + 5 = 873 \\ d &= 49(124) + 25 = 6101 \\ n &= (873(6101) - 1)/124 = 42953 \end{aligned}$$

Hence our public key is $(42953, 873)$, and our private key is $(42953, 6101)$. \triangleleft

Keys in hand, we now need to actually encrypt a message – the power and allure of RSA and this easier variant is that the encryption function is mathematically simple.

Algorithm 11.3 (Kid RSA Encryption). Suppose (n, e) and (n, d) are respectively public and private Kid RSA keys. If m is an integer representing a single character of a plaintext message, then the associated encrypted integer is

$$c = (me) \bmod n.$$

Likewise, if c is an encrypted integer, then the associated decrypted integer is

$$d = (cd) \bmod n.$$

It should be clear that since $(0x) \bmod n = 0$ for every pair of n, x , we should avoid coding any letter as the integer 0.

Example 11.4. Continuing our previous example, we have public key $(42953, 873)$. Suppose we remove punctuation and spaces from the message:

If I have seen further it is by standing on the shoulders of giants².

We will convert to a list of integers by simply using the Python `ord` command.

```
73 70 73 72 65 86 69 83 69 69 78 70 85 82 84 72 69 82 73 84
73 83 66 89 83 84 65 78 68 73 78 71 79 78 84 72 69 83 72 79
85 76 68 69 82 83 79 70 71 73 65 78 84 83
```

Applying the Kid RSA algorithm, we obtain

```
20776 18157 20776 19903 13792 32125 17284 29506 17284 17284
25141 18157 31252 28633 30379 19903 17284 28633 20776 30379
20776 29506 14665 34744 29506 30379 13792 25141 16411 20776
25141 19030 26014 25141 30379 19903 17284 29506 19903 26014
31252 23395 16411 17284 28633 29506 26014 18157 19030 20776
13792 25141 30379 29506
```

◁

Breaking Kid RSA

Unfortunately, it is number-theoretically trivial to break Kid RSA. Consider the values n , d , and e used by the encryption. It must be the case that

$$((me) \bmod n)d \bmod n = m,$$

but then e and d must be multiplicative inverses modulo n . A number $e \in \mathbb{Z}_n$ only has a multiplicative inverse when $\gcd(e, n) = 1$. This leads, however, to an efficient expansion of Euclid's algorithm for the Greatest Common Divisor to recover the value of d .

Theorem 11.5. *Suppose a, b are positive integers, such that $a = bq + r$ with $0 \leq r < b$. Then $\gcd(a, b) = \gcd(b, r)$.*

The application of this theorem repeatedly is called Euclid's algorithm: given positive integers a and b , there are unique q, r with $0 \leq r < b$ such that $a = bq + r$. If $r = 0$, then $\gcd(a, b) = b$. If $r \neq 0$, then $\gcd(a, b) = \gcd(b, r)$.

Algorithm 11.6 (Extended Euclidean Algorithm). Let $s > t$ be positive integers.

1. Set $\mathbf{a} = [s]$, $\mathbf{b} = [t]$, $\mathbf{q} = [\mathbf{a}/\mathbf{b}]$, $\mathbf{r} = [\mathbf{a}\% \mathbf{b}]$, and $\mathbf{p} = [0, 1]$.
2. While the last element of \mathbf{r} is nonzero:

²Issac Newton in a letter to Robert Hooke, 1676

- a. Append the last element of \mathbf{b} to \mathbf{a} .
 - b. Append the last element of \mathbf{r} to \mathbf{b} .
 - c. Append the quotient of those values to \mathbf{q} .
 - d. Append the remainder of those values to \mathbf{r} .
 - e. Append to \mathbf{p} the value $(\mathbf{p}[-2] + \mathbf{p}[-1] * \mathbf{q}[-2]) \% \mathbf{s}$.
3. When the loop terminates, $\mathbf{p}[-1]$ is t^{-1} in \mathbb{Z}_s .

Example 11.7. Applying the algorithm for our public key $(42953, 873)$, we have:

	$p_{-1} = 0$
$42953 = 49(873) + 176$	$p_0 = 1$
$873 = 4(176) + 169$	$p_1 = (0 - 1(49)) \bmod 42953 = 42904$
$176 = 1(169) + 7$	$p_2 = (1 - 42904(4)) \bmod 42953 = 197$
$169 = 24(7) + 1$	$p_3 = (42904 - 197(1)) \bmod 42953 = 42707$
$7 = 7(1) + 0$	$p_4 = (197 - 42707(24)) \bmod 42953 = 6101$

But we had $d = 6101$, so our decryption is broken. Applying

$$f(c) = (6101 \cdot c) \bmod 42953$$

to each integer c in the list of encrypted integers, we recover the original list. \triangleleft

11.3 Plain RSA

Plain RSA follows the same pattern: key generation and distribution, encryption, and decryption. Once again, the public key pair is to be distributed openly, and the private key is never to be distributed.

Algorithm 11.8 (RSA Key Generation).

1. Choose two primes p and q , which for purpose of security should be
 - a. chosen randomly
 - b. similar in magnitude,
 - c. but different in length.
2. Let $n = pq$, the modulus.
3. Compute $\phi(n) = (p - 1)(q - 1) = n - (p + q - 1)$. This is *Euler's totient function*.
4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$.
5. Let $d = e^{-1} \bmod \phi(n)$.

The public key is (n, e) and the private key is (n, d) .

Algorithm 11.9 (RSA Encryption). Suppose (n, e) is an RSA public key and (n, d) is the associated private key. For each integer m in an encoded message, the encrypted integer is

$$c(m) = m^e \bmod n$$

and the decrypted message of a given c is

$$m(c) = c^d \bmod n.$$

Reasonably secure RSA encryption requires the use of large primes – requiring that the minimum bit-length be 1024 is not too small for security’s sake. This in turn leads to very large n . For example, using SageMathCloud for its mathematical strengths over plain Python, I can generate a public key like so in a matter of moments. Note that the first nine lines are all part of the modulus of the key; n in this example is a 2048-bit integer.

```
(20402086648568173481955585730646286137315210218839346974138749941
645259140495567770834936874243791372473116188864372884738105135450
914873426313184766644920305916448220578083159085234843109165718821
649816351354956181220389757990790051050071041390123552155032886914
153510815090479150188398896673272304920300347717062622306615186215
221390731247779732442600196675172980756129027793257452061157450306
999338021825818915796490401265682040597421747260118459364506747473
696770232104151315624298201601728269794649660724878710164849383257
954760868276197510430226341671964363302816497178476763126391107078
274240530722153356995721, 735866671)
```


Part IV

Algorithmic graph theory

Lesson 12

Introduction to Graph Theory

Goals

1. Define a *graph*.
2. Discuss elementary properties of and definitions pertaining to graphs.
3. Discuss several different methods for representing graphs computationally.

12.1 Graphs

Definition 12.1. A *graph* is an ordered pair $\Gamma = (V, E)$, where V is a set of *vertices* and E is a set of *edges*, where edges are unordered pairs of distinct vertices. That is,

$$E \subseteq \{\{u, v\} : u, v \in V\}.$$

In order to simplify notation, we will write uv instead of $\{u, v\}$ for an edge.

Remark. There are many, many different formulations for the definition of a graph. Some definitions are sufficiently relaxed that an edge can loop back from a vertex to itself – these are generally called *loops*. Likewise, edges can be defined as independent objects with *endpoints*, which allows for more than one edge to have vertices u and v as endpoints. I make the distinction that graphs which have loops or multiple edges are *multigraphs*. Some mathematicians are even more relaxed: their edges can contain more than two vertices, and the resulting structure is a *hypergraph*!

Basic terminology

Definition 12.2. Two vertices $v_1, v_2 \in V(\Gamma)$ are *adjacent* if and only if $v_1v_2 \in E(\Gamma)$. An edge $e \in E(\Gamma)$ is *incident* with a vertex $v_1 \in V(\Gamma)$ if and only if there is another vertex $v_2 \in V(\Gamma)$ such that $v_1v_2 = e$.

Definition 12.3. The *degree* of a vertex $v \in V(\Gamma)$ is the number of vertices adjacent to v ; that is,

$$d(v) = |\{u \in V : uv \in E\}|.$$

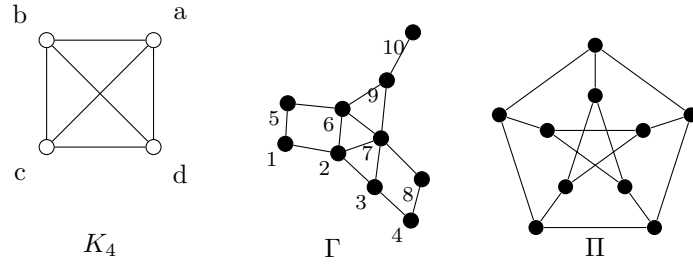


Figure 12.1: Three graphs: K_4 is the *complete graph* on 4 vertices and Π is the *Petersen graph*.

Definition 12.4. A graph $\Gamma_1 = (V_1, E_1)$ is a *subgraph* of $\Gamma = (V, E)$ if and only if $V_1 \subseteq V$ and $E_1 \subseteq E$. Γ_1 is the *subgraph of Γ induced by V_1* if and only if $E_1 = \{u_1v_1 \in E : u_1, v_1 \in V_1\}$. That is, Γ_1 is an induced subgraph of Γ if and only if every edge of Γ between vertices of Γ_1 is an edge of Γ_1 . There are many competing notations for induced subgraphs, but we will denote the induced subgraph of V_1 in Γ by $\langle V_1 \rangle_\Gamma$.

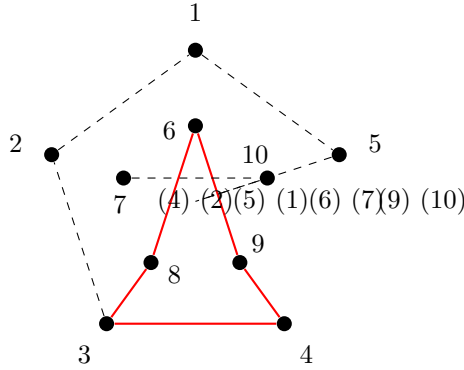


Figure 12.2: The Petersen graph Π and the subgraph induced by $V_1 = \{v_3, v_4, v_6, v_8, v_9\}$. The edge set of $\langle V_1 \rangle_\Pi$ is $E_1 = \{v_3v_4, v_3v_8, v_4v_8, v_6v_8, v_6v_9\}$, highlighted in red.

Definition 12.5. Two graphs $\Gamma = (V, E)$ and $\Gamma' = (V', E')$ are *isomorphic* if there is a function $\phi : V \rightarrow V'$ such that the following three conditions hold.

1. (Injective) If $v_1, v_2 \in V$ with $v_1 \neq v_2$, then $\phi(v_1) \neq \phi(v_2)$.
2. (Surjective) If $v' \in V'$, then there is some $v \in V$ such that $\phi(v) = v'$.
3. (Graph Homomorphism) $v_1v_2 \in E$ if and only if $v'_1v'_2 \in E'$, where $\phi(v_1) = v'_1$ and $\phi(v_2) = v'_2$.

If such a function exists, we write $\Gamma \cong \Gamma'$.

Each of these conditions independently are important, but their combination is essential in graph theory: a graph isomorphism is an *adjacency-preserving bijection between vertex sets*.

Definition 12.6. Let $\Gamma = (V, E)$. A permutation $\phi : V \rightarrow V$ which satisfies condition (3) of Definition 12.5 is a *graph automorphism*. The set of all automorphisms of a graph Γ is denoted $\text{Aut } \Gamma$.

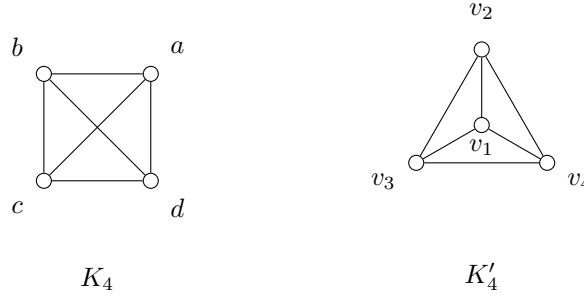


Figure 12.3: While $K_4 \neq K'_4$, the two graphs are isomorphic. In fact, since these are *complete* graphs, every bijection between the sets $\{a, b, c, d\}$ and $\{v_1, v_2, v_3, v_4\}$ is an isomorphism.

Definition 12.7. Suppose $\Gamma = (V, E)$ is a graph and $u, v \in V$. A *path* between u and v is any sequence

$$(u = v_0, e_1, v_1, e_2, v_2, \dots, v_{n-1}, e_n, v_n = v)$$

such that $e_i = v_{i-1}v_i$ for each $i \in \{1, 2, \dots, n\}$ and $v_i \neq v_j$ if $i \neq j$. This can also be called a *u, v -path*. A path containing n edges is a path of *length* n .

Definition 12.8. Let $\Gamma = (V, E)$ be a graph and $v_1, v_2 \in V$. The *distance from v_1 to v_2* is the length of a shortest path between v_1 and v_2 . If no such path exists, the distance from v_1 to v_2 is ∞ . Distance between v_1 and v_2 in the graph Γ is sometimes denoted $d_\Gamma(v_1, v_2)$, which can be easily confused with the notation for degree.

Definition 12.9. A *cycle* is a sequence

$$(v_0, e_1, v_1, e_2, v_2, \dots, v_{n-1}, e_n, v_n = v_0)$$

such that $e_i = v_{i-1}v_i$ for each $i \in \{1, 2, \dots, n\}$ and $v_i \neq v_j$ if $i \neq j$ except for $v_0 = v_n$. A cycle containing n edges is an *n -cycle*, and is isomorphic to the cycle graph $C_n = (\mathbb{Z}_n, \{ab : a, b \in \mathbb{Z}_n, a - b \equiv 1 \pmod{n}\})$.

Definition 12.10. The graph Γ is *connected* if and only if for any two distinct vertices $v_1, v_2 \in V$ there is at least one v_1, v_2 -path.

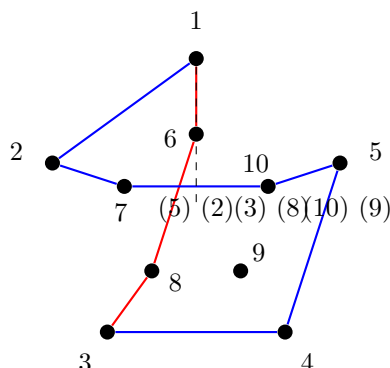


Figure 12.4: The Petersen graph with two distinct (and openly disjoint) v_1, v_3 -paths, one highlighted in red and the other in blue. The union of the red and blue paths is a 9-cycle in Π .

Definition 12.11. A subgraph $\Gamma_1 = (V_1, E_1)$ of a graph $\Gamma = (V, E)$ is a *component* of Γ if and only if $\Gamma_1 = \langle V_1 \rangle_\Gamma$ and $\langle V_1 \cup \{u\} \rangle_\Gamma$ is disconnected for any vertex $u \in V \setminus V_1$. Clearly, a connected graph has only one component.

Definition 12.12. A connected graph containing no cycles is called a *tree*. An arbitrary graph containing no cycles is called a *forest*. For a given graph $\Gamma = (V, E)$, a forest $\Phi = (V, E')$ where $E' \subseteq E$ is called a *spanning forest*. A *spanning tree* of a connected graph is defined analogously.

These are just a few of the “basic definitions” of graph theory. It is a subject which is very appealing for research as the problems are often visually interesting. Since problems in the field are very accessible, some mathematicians are mildly derogatory towards graph theory, calling the field “recreational mathematics.” If that is so, then the vast number of graph theorists are perhaps the luckiest of all mathematicians: their chosen field of research is seen to be fun and games by their colleagues!

All joking aside, graph theory and the larger discipline of combinatorics are deeply applicable fields. There are many “real world” problems which are modeled by discrete systems (rather than continuous systems such as used in differential equations or traditional applied math courses), and graph theory techniques are often the best solution to these problems. So while combinatorics is not generally considered part of applied mathematics, it is very *applicable* mathematics.

12.2 Representing graphs computationally

There are many problems in discrete modeling which are best solved using graph theory, so it is important to be able to represent graphs as some sort of data structure. In fact, there is an abstract data structure called a graph! This data structure is often of limited use when trying to solve mathematical problems on graphs, so we will focus instead on other methods of implementing graphs.

We will begin with a representation that is easily understandable and build towards representations that provide more theoretical power.

Edge lists

The simplest way to represent a graph is to provide a list of the vertices and the edges; while the edges can be stored either as lists or tuples, we will prefer tuples so they are immutable. We'll consider again the Petersen graph, this time with vertices labeled from 0 through 9.

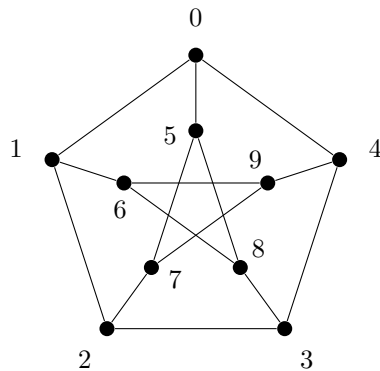


Figure 12.5: The Petersen graph with vertices $V = \{v_0, v_1, \dots, v_9\}$.

In Python, the list of edges can be given using the integer indices of vertices as the “name” of the vertex. So the vertex list and list of edges as tuples are as given below:

```
V = list(range(10))
E = [(0, 1), (0, 4), (0, 5), (1, 2), (1, 6), (2, 3), (2, 7), (3, 4), (3, 8),
      (4, 9), (5, 7), (5, 8), (6, 8), (6, 9), (7, 9)]
```

We notice that each edge is listed only once, in one direction – algorithms must take this into account! Even though v_0v_1 is the same edge as v_1v_0 , there are often times when we will want to use the edge in the latter order rather than the former. This makes an edge list slightly less useful than some of the other representations when we move into implementing algorithms.

Equivalent to an edge list is an edge dictionary: the dictionary representation of this edge list is

```
d = {0: [1, 4, 5], 1: [2, 6], 2: [3, 7], 3: [4, 8], 4: [9], 5: [7, 8], 6: [8, 9],
      7: [9]}
```

This suffers from exactly the same limitation as the edge list. A simple solution is to use a “double-ended” dictionary:

```
ded = {0: [1, 4, 5], 1: [0, 2, 6], 2: [1, 3, 7], 3: [2, 4, 8], 4: [0, 3, 9],
      5: [0, 7, 8], 6: [1, 8, 9], 7: [2, 5, 9], 8: [3, 5, 6], 9: [4, 6, 7]}
```

While this stores every edge twice, it allows us to access each edge from either endpoint.

Adjacency matrices

From the idea of a double-ended edge dictionary it is a small step to build the adjacency matrix of a graph.

Definition 12.13. Suppose $\Gamma = (V, E)$ is a graph with vertices $V = \{v_0, v_1, \dots, v_{n-1}\}$. Then the *adjacency matrix* of Γ is the $n \times n$ matrix $A = [a_{i,j}]$ with

$$a_{i,j} = \begin{cases} 1 & v_i v_j \in E \\ 0 & v_i v_j \notin E \end{cases}.$$

The adjacency matrix of the Petersen graph as above is

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

. If we have the double-ended dictionary stored as `ded` and the vertex list as `V`, we can create this adjacency matrix easily using a list comprehension:

```
A = [[1 if v in ded[u] else 0 for v in V] for u in V]
```

Lesson 13

Shortest Paths

Goals

1. Discuss the shortest path problem.
2. Explain Dijkstra's shortest path algorithm.
3. Implement Dijkstra's algorithm.

13.1 Erdős Number

The mathematician Paul Erdős is renowned for having been one of the most prolific mathematicians of the 20th century, as well as one of the most social: he wrote papers with more than 500 collaborators over his career. Due to his frequent collaborations, one of his coauthors assigned to Paul an “Erdős number” of 0, for being Paul Erdős. Each of Erdős's coauthors were assigned an Erdős number of 1. Their coauthors who did not coauthor with Erdős were assigned an Erdős number of 2, and so on.

Essentially, this builds a *rooted collaboration graph*: a graph whose vertices are mathematicians and edges are coauthorship on a paper, with a special *root vertex* designated as Paul Erdős. The distance in this graph is the length of a shortest path between vertices. The question then is how to determine a shortest path.

While this problem is described in terms of something trivial, shortest paths are important in other mediums as well. If a transmission of a message in a network is more likely to have errors introduced with each subsequent connection traversed, it is important to minimize the number of transmissions. With a little creativity, the shortest path problem can be applied to hyperspace routing for a particular type of fictional faster-than-light travel.

13.2 Dijkstra's Algorithm

In 1956, Dutch computer scientist Edsger Dijkstra developed an algorithm for solving this problem. His solution, now known as Dijkstra's algorithm, is both elegant and simple.

Algorithm 13.1 (Dijkstra's Shortest Path Algorithm). Let $G = (V, E)$ be a

graph, and let x be a particular vertex in V . We will define $d(v)$ to be the currently known distance from x to v , and $V' = \{\}$ to be an initially empty set. For each $v \in V \setminus \{x\}$, we define $p(v)$ to be the *predecessor* of v in a shortest path from x .

1. Mark $d(x) = 0$ and $d(v) = \infty$ for every $v \in V \setminus \{x\}$.
2. While $V \neq V'$, do the following:
 - a. Let u be any vertex in $\{u \in V \setminus V' : d(u) = \min \{d(v) : v \in V \setminus V'\}\}$.
 - b. Add u to V' .
 - c. For each neighbor v of u , if $d(v) > d(u) + 1$, set $d(v) = d(u) + 1$, and set $p(v) = u$.
3. Return the triple $(v, d(v), p(v))$ for each $v \in V$.

In fact, Dijkstra's algorithm produces what is called a *breadth-first traversal* of Γ from the root vertex x . For each vertex v in the same component of Γ as v_0 , there is a finite sequence of predecessors which can be followed backwards from v to x ; for vertices in other components, no such predecessor exists and the distance $d(x, v)$ is infinite. We'll illustrate the algorithm with the example of a graph with two components, one of which is a single isolated vertex.

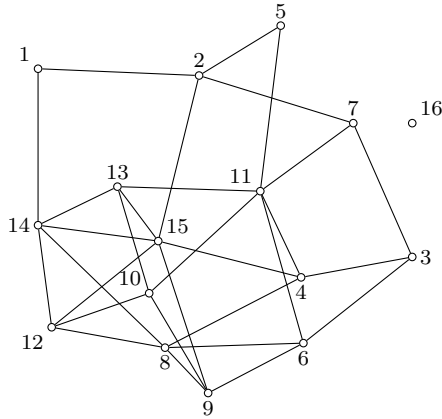


Figure 13.1: A graph Γ with two components: the vertex v_{16} is isolated from all other vertices.

Example 13.2. Consider the graph $\Gamma = (V, E)$ with vertex set $V = \{1, 2, \dots, 16\}$ and edge set

$$E = \{(1, 2), (1, 14), (2, 5), (2, 7), (2, 15), (3, 4), (3, 6), (3, 7), (4, 8), (4, 11), (4, 15), \\ (5, 11), (6, 8), (6, 9), (6, 11), (7, 11), (8, 9), (8, 12), (8, 14), (9, 10), (9, 15), \\ (10, 11), (10, 12), (10, 13), (11, 13), (12, 14), (12, 15), (13, 14), (13, 15), (14, 15)\}.$$

This graph is depicted in Figure 13.1. If we designate the root vertex to be $x = v_1$, we can begin the process. The iterations are explained briefly, and all steps are tabulated below in Table 13.1. Since $x = v_1$, we set

$$d(v) = \begin{cases} \infty & v \neq v_1 \\ 0 & v = v_1 \end{cases}$$

and $p(v)$ takes no values. As the only unvisited vertex at finite distance from x , we mark v_1 as visited by putting $V' = \{v_1\}$. Looking at the neighbors of v_1 , we see that $N(v_1) = \{v_2, v_{14}\}$; since the distance to both is currently infinite, we redefine d and p as follows:

$$d(v) = \begin{cases} 0 & v = v_1 \\ 1 & v \in \{v_2, v_{14}\} \\ \infty & \text{otherwise} \end{cases} \quad p(v) = 1 \text{ for } v \in \{v_2, v_{14}\}.$$

Since $2 < 14$, we choose next to visit v_2 ; hence $V' = \{v_1, v_2\}$ and we observe that the as-yet-unvisited neighbors of v_2 are v_5, v_7 , and v_{15} . We again update d and p :

$$d(v) = \begin{cases} 0 & v = v_1 \\ 1 & v \in \{v_2, v_{14}\} \\ 2 & v \in \{v_5, v_7, v_{15}\} \\ \infty & \text{otherwise} \end{cases} \quad p(v) = \begin{cases} 1 & v \in \{v_2, v_{14}\} \\ 2 & v \in \{v_5, v_7, v_{15}\} \end{cases}$$

Proceeding in this manner, we will visit vertices $v_{14}, v_5, v_7, v_{15}, v_8$, and v_{12} before something strange happens: every vertex will have been visited or “seen” on its shortest path, except for v_{16} . While v_{16} will eventually be added to V' , it will never be seen as a neighbor of another vertex, and will never be assigned a predecessor. \triangleleft

Step	Current Vertex	Distance	Predecessor	Updating vertices $\{v_j : j \in J\} : (d(v), p(v))$
1	1	0	–	$\{2, 14\} : (1, 1)$
2	2	1	1	$\{5, 7, 14\} : (2, 2)$
3	14	1	1	$\{8, 12, 13\} : (2, 14)$
4	5	2	2	$\{11\} : (3, 5)$
5	7	2	2	$\{3\} : (3, 7)$
6	15	2	2	$\{2, 9\} : (3, 15)$
7	8	2	14	$\{6\} : (3, 8)$
8	12	2	14	$\{10\} : (3, 12)$
9	13	2	14	
10	11	3	5	
11	3	3	7	
12	6	3	8	
13	10	3	12	
14	4	3	15	
15	9	3	15	
16	16	∞	–	

Table 13.1: Iterations of Dijkstra's algorithm on the graph Γ from Figure 13.1.

Are there other ways to find shortest paths? Of course. The beauty of using Dijkstra's algorithm is that when you first encounter a vertex, you're already recording its shortest path. It will never be the case that this algorithm encounters a vertex at some large distance and then updates it to a smaller distance.

13.3 Implementing Dijkstra's algorithm

The only really sophisticated use of data structures in the implementation we will use of Dijkstra's algorithm is that we will use a double-ended dictionary of edges: that is, rather than only recording that v_i is adjacent to v_j when $j > i$, we will record both ends of every edge. As the edge dictionary which is provided is not necessarily double-ended, the first step of our implementation will be to create it.

```
def double_dict(edge_dict):
    """ edge_dict should be a dictionary with integer keys and list of
    integer values. double_dict will return dictionary and vertex set of
    the graph."""
    m = min(edge_dict.keys())          # These set up to return the
    M = max(edge_dict.keys())          # vertex set of the graph.
    out_dict = {}
    for u,nbrs in edge_dict.items():
        for v in nbrs:
            m = min(v, m)              # Finding the min
            M = max(v, M)              # Finding the max
            try:
                out_dict[u] += [v]      # Updating double-endedness
            except KeyError:
                out_dict[u] = [v]
            try:
                out_dict[v] += [u]      # Updating double-endedness
            except KeyError:
                out_dict[v] = [u]
    return out_dict, [m+i for i in range(M-m+1)] # return dict and vertices
```

Using these functions makes the implementation of Dijkstra's algorithm almost natural.

```
def dijkstra(edge_dict, root, **kwargs):
    dbl_dic, V = double_dict(edge_dict)
    order = len(V)
    d = {v:(order+1 if v!=root else 0) for v in V} # Setup distance mapping
    p = {v:None for v in V}                       # Setup predecessors
    vis = []
    while len(vis)<len(V):
        dpu = sorted((dist,p[v],v) for v,dist in d.items() if v not in vis)
        cd, cp, cu = dpu[0]                       # Find nearest unvisited
        vis += [cu]
        if cu in dbl_dic.keys():
            for v in dbl_dic[cu]:
                if v not in vis and d[v]>cd+1:
                    d[v] = cd+1                   # Update anything nearer
                    p[v] = cu                     # than previously seen.
    return d,p
```

Lesson 14

Minimum spanning trees

Goals

1. Explore the minimum spanning tree problem
2. Explain Kruskal's algorithm
3. Implement Kruskal's algorithm

14.1 Weighted graphs & minimum spanning trees

A natural extension of graph theory is to consider a graph associated with a *weight function*.

Definition 14.1. Suppose $\Gamma = (V, E)$ is a graph with vertex set V and edge set E . A *weight function* on Γ is a function $w : E \rightarrow \mathbb{R}$. Such a weighted graph will be denoted $\Gamma = (V, E, w)$.

Often the weight function somehow represents the “cost” of traversing an edge – in this context it becomes almost obvious why finding a spanning tree of minimum weight would be useful. Recall that a *spanning forest* for a graph is an acyclic subgraph covering all vertices; a *spanning tree* is a connected acyclic subgraph covering all vertices.

A minimum spanning tree provides a way to connect all the vertices in a graph in a non-redundant way with the least associated cost.

14.2 Kruskal's algorithm

Joseph Kruskal first published his solution to the minimum spanning tree problem in 1956; again, this is a wonderful, almost intuitive algorithm, which hinges on an awesome use of mathematical induction. The caveat is that the weight function for Kruskal's algorithm must be a non-negative function, $w : E \rightarrow \{x \geq 0 : x \in \mathbb{R}\}$.

Theorem 14.2. If $\Gamma = (V, E)$ is a forest and $uv \in E$ is an edge where u and v are not in the same component of Γ , then $\Gamma' = (V, E \cup \{uv\})$ is a forest.

Proof. Suppose $\Gamma = (V, \{ \})$ is a totally disconnected graph, and let $u_0, u_1 \in V$. Then u_0 and u_1 are (naturally) in separate components. The graph $\Gamma' = (V, \{u_0u_1\})$ contains no cycles, since a graph with only one edge cannot contain a cycle.

Suppose $\Gamma = (V, E)$ is a forest and $u_0, u_1 \in V$ are vertices in different components of Γ . Assume, towards a contradiction, that $\Gamma = (V, E \cup \{u_0u_1\})$ contains a cycle. Then there is a cycle $C = (v_0, e_1, v_1, e_2, v_2, \dots, v_{n-1}, e_n, v_n = v_0)$ where $v_i \in V$ and $v_i \neq v_j$ when $i \neq j$ and $\{i, j\} \neq \{0, n\}$. If $u_0u_1 \neq e_j$ for any $j \in \{1, 2, \dots, n\}$, then we have contradicted the assumption that Γ was a forest. On the other hand, if $u_0u_1 \in \{e_1, e_2, \dots, e_n\}$, then we may relabel the cycle $C = (v_0 = u_0, e_1, v_1, e_2, v_2, \dots, v_{n-1} = u_1, e_n = u_0u_1, v_n = u_0 = v_0)$. But then $(v_0 = u_0, e_1, v_1, e_2, v_2, \dots, v_{n-1} = u_1)$ is a u_0, u_1 -path, contradicting the assumption that u_0 and u_1 are in different components of Γ . In either case, we have a contradiction. \square

Hence we have an immediate method of producing a spanning forest – include any edge which does not connect components. More importantly, we have all that is necessary for Kruskal's algorithm: if the graph is weighted, the edge of minimal weight which does not introduce a cycle is added in every iteration.

Algorithm 14.3 (Kruskal's Spanning Tree algorithm). Let $\Gamma = (V, E, w)$ be a weighted graph where $V = \{v_1, v_2, \dots, v_n\}$ and $E = \{e_1, e_2, \dots, e_m\}$ such that $w(e_i) \leq w(e_j)$ whenever $i < j$.

1. Suppose $C_1 = \{v_1\}$, $C_2 = \{v_2\}$, \dots , $C_n = \{v_n\}$, and $E_0 = \{ \}$
2. For each $j = 1, 2, \dots, m$, let $uv = e_j$. Do the following:
 - a. If $u \in C_k$ and $v \in C_\ell$ and $C_k \neq C_\ell$ where $k < \ell$, then let $C_k = C_k \cup C_\ell$ and let $E_j = E_{j-1} \cup \{e_j\}$. Otherwise, let $E_j = E_{j-1}$.
3. $\Gamma' = (V, E_m)$ is a minimum spanning tree for Γ .

Example 14.4. Consider the list of (w, u, v) triples below, where w is the weight assigned to the edge uv :

(4, 'a', 'b')	(14, 'a', 'f')	(18, 'a', 'h')	(7, 'a', 'i')
(5, 'a', 'j')	(1, 'a', 'l')	(4, 'a', 'o')	(5, 'b', 'i')
(8, 'b', 'k')	(10, 'b', 'l')	(11, 'b', 'm')	(9, 'c', 'e')
(5, 'c', 'f')	(16, 'c', 'g')	(6, 'c', 'j')	(4, 'c', 'm')
(2, 'c', 'o')	(11, 'd', 'e')	(18, 'd', 'h')	(10, 'd', 'j')
(2, 'e', 'h')	(9, 'e', 'k')	(3, 'e', 'o')	(19, 'f', 'k')
(12, 'h', 'i')	(2, 'h', 'j')	(15, 'i', 'k')	(3, 'i', 'l')
(4, 'j', 'l')	(16, 'n', 'o')		

The graph Γ with this edge set on $V = \{a, b, c, \dots, o\}$ and weight function w is depicted in Figure 14.1. To begin applying Kruskal's algorithm to find a spanning tree of Γ with minimal weight, we assign each vertex to its own component; thus, the partition of the vertices is

$$\mathcal{C} = \{C_0 = \{a\}, C_1 = \{b\}, C_2 = \{c\}, \dots, C_{14} = \{o\}\}.$$

We begin with the lightest edge, and check each edge to determine whether it connects components. If not, the edge is included in the spanning tree.



Let $C(v) = k$ if and only if $v \in C_k$. The “lightest” edge is an al -edge, with weight 1. Since $C(a) = 0$ and $C(l) = 11$, we include the edge al . Hence we update our partition, with the new partition $C_0 = \{a, l\}$ and $C_{11} = \{\}$. The next lightest edge is co , with weight 2. Since $C(c) = 2$ and $C(o) = 14$, we again include the edge co . Updating the partition, we now have $C_2 = \{c, o\}$ and $C_{14} = \{\}$. The next two edges are similar: firstly $w(eh) = 2$, $C(e) = 4$, and $C(h) = 7$, so we include eh and put $C_4 = \{e, h\}$ and $C_7 = \{\}$; secondly $w(hj) = 2$, $C(h) = 4$ from the previous update, and $C(j) = 9$, so we include hj and put $C_4 = \{e, h, j\}$ and $C_9 = \{\}$. The next iteration shows us $w(eo) = 3$, but now we merge two cells of the partition of which neither is a single-element cell. As the algorithm is written naively¹, we simply merge the cells into that of the vertex which comes first in the vertex order. So since $C(e) = 4$ and $C(o) = 2$, we include eo and obtain $C_4 = \{c, e, h, j, o\}$ and $C_2 = \{\}$.

¹This is naive, because a better approach would involve merging the smaller cell into the larger one – this would require fewer assignments. Kruskal’s algorithm is a good place to start thinking about “smart” data structures, which improve the computational complexity of the algorithm by reducing the number of loops.

9	c	8	e	8	False
9	e	8	k	8	False
10	b	8	l	8	False
10	d	3	j	8	True
11	b	3	m	3	False
11	d	3	e	3	False
12	h	3	i	3	False
14	a	3	f	3	False
15	i	3	k	3	False
16	c	3	g	6	True
16	n	13	o	3	True
18	a	13	h	13	False
18	d	13	h	13	False
19	f	13	k	13	False

The set of included edges is $E' = \{ab, al, ao, bk, cf, cg, cm, co, dj, eh, eo, hj, il, no\}$. The result of this process is depicted in Figure 14.2. \triangleleft

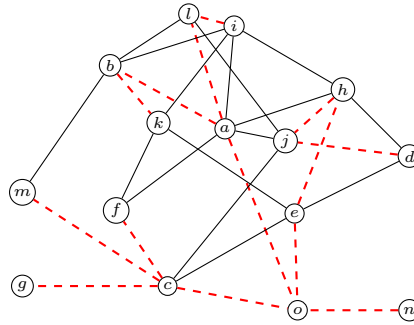


Figure 14.2: The same graph Γ from Figure 14.1 with the minimum spanning tree discovered via Kruskal's algorithm highlighted with red dashed edges.

14.3 Implementing Kruskal's algorithm with “smart data structures”

In a very naive approach to Kruskal's algorithm, the function $C(v)$ reporting the component of vertex v is not implemented. This requires two searches through the components for each edge. Since we will implement the components as a list of lists of vertices, we may (at the cost of a small amount of memory) implement the function $C : V \rightarrow \mathbb{N}$ as a dictionary storing the index of the component containing each vertex $v \in V$. The variable `comps` stores the actual list of components.

```
def smart_kruskal(wedge_dict):
    v = []
    wedges = []
    for u, nbrs in wedge_dict.items():
        wedges += [(w, u, v) for v, w in sorted(nbrs.items())]
```



```
    V += ([u]+[v for v in nbrs.keys() if v not in V])
V.sort()
wedges.sort()
C = {v:i for i,v in enumerate(V)}
comps = {i:[v] for i,v in enumerate(V)}
included = []
excluded = []
for w,u,v in wedges:
    cu = C[u]
    cv = C[v]
    if cu == cv:
        excluded += [(u,v,w)]
    else:
        included += [(u,v,w)]
        C[v] = cu
        for vert in comps[cv]:
            comps[cu].append(vert)
            C[vert]=cu
return [included,excluded]
```

Lesson 15

Maximum flow in a capacitated network

Goals

1. Define a *capacitated network*.
2. Introduce the maximum flow and minimum cut problems and the max-flow, min-cut theorem.
3. Introduce Dinitz' Algorithm to solve the max-flow/min-cut problem.

15.1 Capacitated networks

All information flows over some sort of network, whether it be electronic or otherwise; you can think of communicating person-to-person as a pair of nodes and a regular graph edge connecting them, while a person lecturing to a group might be modeled more accurately by a hypergraph¹. Similarly, the logistic problem of transporting goods and services across the country can be modeled using a transportation network.

In both of these cases, there is a sort of maximum capacity for the transfer, and this gives us the concept of a *capacitated network*. This is much like a weighted graph, but we tend to think of networks as directed. Additionally we must consider that not every connection in a network can be utilized to its full capacity at all times; hence we have two numbers associated with each edge, the *flow* and the *capacity*.

A capacitated network also has two special vertices: the *source*, s , and *target* (or sink), t . The flow begins from the source and terminates at the sink – a flow is valid if and only if the flow into each vertex is equal to the flow out of the vertex, with exceptions at s and t , and the flow across each directed edge is less than or equal to the capacity of that edge.

15.2 Max flow

In nearly every application, the desire is to maximize the flow from s to t across the network; since what is flowing across most networks is not a physical fluid, the problem of maximiz-

¹Remember that a *hypergraph* permits edges to contain more than two vertices.

ing flow is discrete-mathematical rather than a problem of physics solved by differential equations.

Definition 15.1. Suppose $\Delta = (V, E, s, t)$ is a directed graph with *source vertex* s and *target vertex* t . A *capacity function* $c : E \rightarrow \mathbb{R}^+$ for Δ is the maximum amount of flow which can pass across an edge; this is denoted $c(u, v)$ for the directed edge $uv \in E$. A *flow function* $f : E \rightarrow \mathbb{R}^+$ for Δ is a function satisfying the following two constraints:

1. (Capacity Constraint) $f(u, v) \leq c(u, v)$ for all $uv \in E$, and
2. (Conservation of Flow) for each $v \in V \setminus \{s, t\}$,

$$\sum_{uv \in E} f(u, v) = \sum_{vu \in E} f(v, u).$$

The *value of flow* for a flow function f is defined by

$$|f| = \sum_{v \in V} f(s, v).$$

Generally, we wish to maximize the value $|f|$ over all possible flows.

Definition 15.2. Suppose $\Delta = (V, E, c, s, t)$ is a capacitated network with source s and target t . An $s - t$ cut is a partition $C = (S, T)$ such that $s \in S$ and $t \in T$. The *cut-set* of C is the set of edges $\{uv \in E : u \in S, v \in T\}$. We notice that if a flow of 0 is put on all edges in the cut-set of C , then $|f| = 0$. The *capacity* of the cut $C = (S, T)$ is

$$c(S, T) = \sum_{\{uv \in E : u \in S, v \in T\}} c(u, v).$$

Just as it is interesting to determine a maximum flow over a network, there are applications when finding a minimum-capacity cut-set is desirable.

Theorem 15.3 (Max-flow Min-cut Theorem). *The maximum value of an s - t flow is equal to the minimum capacity of an s - t cut.*

15.3 Dinitz' Algorithm

The original solution to the problem of calculating a maximum flow was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. The modified version presented here was produced by Yefim Dinitz as a "response to a pre-class exercise in Adel'son Vel'sky's Algorithm class." According to Dinitz, he was at the time unaware of the basic details of the Ford-Fulkerson algorithm upon which his algorithm improves.

Both algorithms proceed in this general fashion: given a flow, determine which edges have "residual capacity." In the residual graph, try to find a flow which can augment the current flow. When no such flow can be found, a maximum flow has been found.

The foremost reason an improvement was needed to the Ford-Fulkerson algorithm is its failure to terminate in certain cases of networks with irrational capacities. As the algorithm is successful in constructing a maximum flow whenever it terminates, an improvement to the algorithm was more likely than a replacement. Dinitz' improvement utilized shortest paths

to overcome the conceptual flaw in Ford-Fulkerson during the step of finding an augmenting path. Hence we will first discuss the modification to Dijkstra's algorithm which produces the necessary "shortest path graph."

The modification to Dijkstra's algorithm to construct a subgraph consisting of all shortest paths from s to t is conceptually deep but easy to implement. Suppose $v \in V$ with $d(s, v) = k$, where s is the source vertex. Rather than recording a single s, v -path, we records every vertex $u \in V$ such that $d(s, u) = k - 1$ and there is an arc² from u to v .

Algorithm 15.4 (All Shortest Paths from s (Modified Dijkstra)). Suppose that $\Delta_f = (V, E_f, c_f, s, t)$ is a capacitated network with source vertex s and target vertex t .

1. Initialize the distance and predecessor functions,

$$d(s, v) = \begin{cases} 0, & v = s \\ \infty, & v \neq s \end{cases} \quad p(v) = \{\}$$

for each $v \in V$. Also, define a set $V' = \{\}$ of visited vertices.

2. While $|V'| < |V|$, repeat the following steps:

- a. Let $u \in V \setminus V'$ be an unvisited vertex nearest to s ; that is,

$$d(s, u) = \min \{d(s, v) : v \in V \setminus V'\}.$$

- b. For each $v \in V \setminus V'$ such that $uv \in E_f$, $d(s, u) + 1 \leq d(s, v)$, and $c_f(u, v) > 0$:

- i. update the distance to v by setting $d(s, v) = d(s, u) + 1$, and
- ii. update the set of predecessors of v , $p(v)$, to include u .

- c. Add u to V' .

3. Return the values of $d(s, v)$ and $p(v)$ for each $v \in V$.

The true secret to Dinitz' algorithm is not in its use of this shortest path algorithm once, but twice: if the output $p(v)$ is used to define a new set of arrows $E_b = \{uv : v \in p(u)\}$ we can use the modified shortest path algorithm now on the graph with edge set E_b and source t to ensure that the only shortest paths we retain are paths from s to t .

Example 15.5. The graph in Figure 15.1(A) is a capacitated network $\Delta = (V, E, c, s, t)$ with $V = \{v_1, v_2, \dots, v_6\}$; only the indices of vertices are shown in the figure. Suppose we wish to apply Dijkstra's algorithm to this graph rooted at v_1 . The neighbors of v_1 are $N(v_1) = \{v_2, v_3\}$; visiting each of these in turn we see that the neighbors of v_2 are $N(v_2) = \{v_3, v_4, v_5\}$; however, $d(v_1, v_3) = 1 < 2 = d(v_1, v_2) + 1$, so we do not update the predecessors of v_3 . Likewise, $N(v_3) = \{v_4, v_5, v_6\}$. As this provides new paths of the same length to both v_4 and v_5 , we add v_3 as a predecessor of each of those vertices. Recalling

²Recall that we are working over directed graphs now.

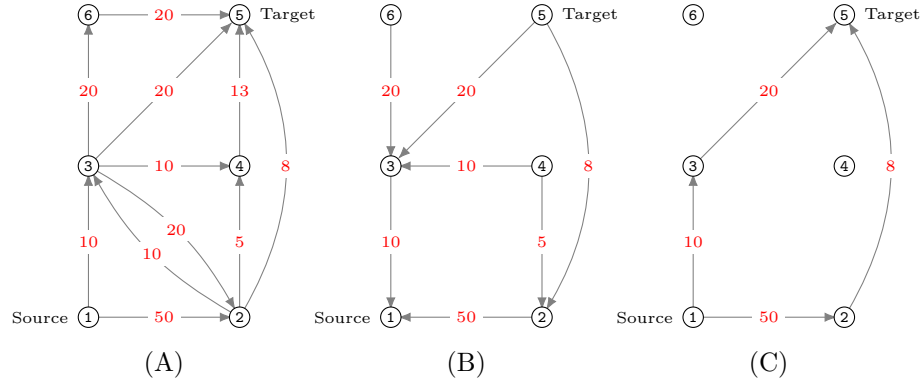


Figure 15.1: (A) A capacitated network, $\Delta = (V, E, c, v_1, v_5)$; (B) the result Δ' of applying Algorithm 15.4 on Δ ; (C) the “layered graph” $\Lambda = (V, E_\ell, c_\ell, v_1, v_5)$ resulting from the application of Algorithm 15.4 on Δ' .

that the modified Dijkstra’s algorithm provides a sort of “predecessor graph,” we have obtained the capacitated network shown in Figure 15.1(B). The second step of producing the layered data structure is to apply Dijkstra’s algorithm on this intermediate graph, rooted now at the target vertex v_5 . As the graph is directed, we see that vertices v_4 and v_6 are unreachable in the intermediate graph when rooted at v_5 ; hence the resulting layered graph of Figure 15.1(C) with vertex v_1 at distance 0, vertices v_2 and v_3 at distance 1, and vertex v_5 at distance 2. \triangleleft

It is important to make a note about the structure of $\Lambda = (V, E_\ell, c_\ell, s, t)$ produced from two successive applications of Algorithm 15.4. We first produced Δ' from Δ by including the reverse edges of exactly those edges lying on shortest paths emanating from s ; we then construct Λ from Δ' by including only those edges on shortest paths terminating at t . Hence *every path in Λ beginning at s will terminate at t and will be a shortest such path*. This structure must be maintained as we augment the flow across the network, for two reasons: as edges become saturated³ they must be removed from the residual capacity graph, just as residual capacities of unsaturated edges must be decreased. When saturated edges are removed, this can result in a vertex with no successors or with no predecessors, and both of these types of vertex lie on no s, t -path in Λ .

The graph Λ is referred to as a layered structure since we can think of the vertices as lying in distance layers away from s such that any edge $uv \in E_\ell$ has the property $d(s, v) = d(s, u) + 1$. Thus maintaining those layers means we need no special algorithm to find augmenting paths: any path emanating from s in a well-maintained layered graph will eventually terminate at t . The order in which paths are followed can change the final assignment of flow to edges, but interestingly cannot change the total final value of the flow. The flow assigned along the path is the minimum residual capacity along any edge of the path.

³An edge uv is *saturated* when $f(u, v) = c(u, v)$, where f is the flow and c is the capacity.

Algorithm 15.6 (Pathfinding). Suppose $\Lambda = (V, E_\ell, c_\ell, s, t)$ is the result of two applications of Algorithm 15.4 on a capacitated network $\Delta = (V, E, c, s, t)$.

1. Let $u_0 = s$.
2. Inductively choose u_{i+1} for $i \geq 0$ such that $u_i u_{i+1} \in E_\ell$, until $u_n = t$.
3. Let $f^* = \min \{c_\ell(u_i, u_{i+1}) : i = 1, 2, \dots, n-1\}$.
4. The augmenting path to be returned is the path $(u_0, u_1, u_2, \dots, u_n)$ and the augmenting flow is f^* on each of those edges.

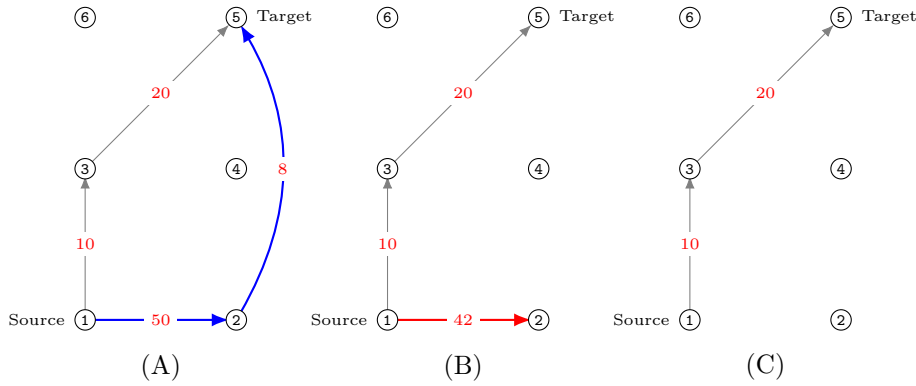


Figure 15.2: (A) An augmenting path with flow value $f^* = 8$ in Λ from Example 15.5. (B) After updating the residual capacities of edges in E_ℓ , the edge $v_2 v_5$ was saturated and therefore removed. This leaves v_2 with no successor vertices, so the edge $v_1 v_2$ (shown with reduced capacity) must also be removed as no additional flow can move from v_1 to v_5 along $v_1 v_2$. (C) The graph Λ after maintenance.

Example 15.7 (Continued from previous example.). In our graph Λ , $u_0 = s = v_1$ has only two successors: v_2 and v_3 . Choosing $u_1 = v_2$, we see that the only successor of u_1 in Λ is $v_5 = t$, so $u_2 = v_5$. Our augmenting path is then (v_1, v_2, v_5) ; as this flow only consists of two edges of capacities 50 and 8 respectively, it is clear that $f^* = \min \{50, 8\} = 8$.

The current flow in our graph is $f(u, v) = 0$ for all $uv \in E$, and we now have an augmenting path and flow. So we update the flow value to accommodate the augmentation:

$$f(u, v) = \begin{cases} 8, & uv \in \{v_1 v_2, v_2 v_5\} \\ 0, & uv \in E \setminus \{v_1 v_2, v_2 v_5\}. \end{cases}$$

We update our residual capacities and see that edge $v_2 v_5$ is now saturated; removing this edge from E_ℓ we find that now v_2 has no successors and so all edges terminating at v_2 must also be removed in graph maintenance.

Having maintained the graph, we see that the path (v_1, v_3, v_5) remains and permits another augmenting flow of $f^* = 10$. Again updating, we have

$$f(u, v) = \begin{cases} 8, & uv \in \{v_1v_2, v_2v_5\} \\ 10, & uv \in \{v_1v_3, v_3v_5\} \\ 0, & uv \in E \setminus \{v_1v_2, v_1v_3, v_2v_5, v_3v_5\}. \end{cases}$$

Now no augmenting flows remain in Λ after maintenance, as v_1v_3 is saturated and hence v_3 has no predecessors in Λ .

In order to continue to augment the flow, we must again determine the residual capacities and build a new residual graph Δ_f taking into account the current flow f . \triangleleft

Remark. About graph maintenance: there are several different ways that this can be handled, all with the same result but different in individual process. Dinitz originally kept track of the endpoints of edges which became saturated as the algorithm updated the residual capacities along an augmenting path. After these capacities were updated, the algorithm processed through lists of vertices which might have been left with either no successors or no predecessors. In our implementation, we opt for a recursive solution: when an edge $uv \in E_\ell$ becomes saturated, we call a process `clean_to_source` on its head vertex u and then a process `clean_to_target` on its tail vertex v . These two processes behave similarly, but in opposite directions. The first, `clean_to_source`, determines whether u has any successors; if not, `clean_to_source` is run on each predecessor w of u . Likewise `clean_to_target` checks whether v has any predecessors, and if not, `clean_to_target` is run on each successor w of v .

Algorithm 15.8 (Maintenance of layered graph). Suppose $\Lambda = (V, E_\ell, c_\ell, s, t)$ is the layered graph structure output from two successive applications of Algorithm 15.4, that $P = \{s = u_0, u_1, \dots, u_{n-1}, u_n = t\}$ is an augmenting path with augmenting flow value f^* .

1. For each $i \in \{0, 1, 2, \dots, n-1\}$, update $c_\ell(u_i, u_{i+1})$ by subtracting from it the value f^* .
2. For an edge $uv \in E_\ell$ where after updating, $c_\ell(u, v) = 0$, do the following:

Clean to source from a vertex u :

- a. The vertex u has no successors in Λ precisely when

$$\{w \in V : c_\ell(u, w) > 0\} = \{\}.$$

- b. If u has no successors and is not the source vertex s , consider the set $P = \{w \in V : c_\ell(w, u) > 0\}$ of predecessors of u .
- c. For each $w \in P$, set $c_\ell(w, u) = 0$. For each $w \in P$ with no successors other than u , **clean to source from the vertex w** .

Clean to target from a vertex v :

- a. The vertex v has no predecessors in Λ precisely when

$$\{w \in V : c_\ell(w, v) > 0\} = \{\}.$$

- b. If v has no successors and is not the target vertex t , consider the set $S = \{w \in V : c_\ell(v, w) > 0\}$ of successors of v .
- c. For each $w \in S$, set $c_\ell(v, w) = 0$. For each $w \in S$ with no predecessors other than v , **clean to target from the vertex w** .
3. After both cleaning processes have finished for all saturated edges, remove from E_ℓ any edge uv with $c_\ell(u, v) = 0$.

We now have all the pieces in place to actually state Dinitz' algorithm.

Algorithm 15.9 (Dinitz' Max Flow Algorithm). Suppose we are given a capacitated network $\Delta = (V, E, c, s, t)$ with capacity function $c : E \rightarrow \mathbb{R}^+$, source s , and target t .

1. Define the initial flow $f(u, v) = 0$ for $uv \in E$ or $vu \in E$.
2. Repeat the following:
 - a. Calculate the *residual capacity function* $c_f : V \times V \rightarrow \mathbb{R}^+$, given by:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v), & uv \in E \\ f(u, v), & vu \in E \end{cases}.$$

We remark that a positive flow of $f(u, v)$ provides exactly that much capacity to the reverse edge vu .

- b. Let $\Delta_f = (V, E_f, c_f, s, t)$ be the *residual graph* with

$$E_f = \{uv \in E : c_f(u, v) > 0\}.$$

- c. Apply Algorithm 15.4 to Δ to produce Δ' ; apply Algorithm 15.4 again to Δ' to obtain the layered graph $\Lambda = (V, E_\ell, c_\ell, s, t)$. **When no shortest paths from s to t are found in constructing Δ' , the flow is optimal.**
 - d. Repeat the following:
 - i. Use Algorithm 15.6 to find an augmenting path P with flow value f^* in Λ . If no augmenting path exists, return to step a.
 - ii. Update f by adding f^* to the flow for each edge $u_i u_{i+1}$ where u_i, u_{i+1} are adjacent vertices in P .
 - iii. Perform Algorithm 15.8 upon Λ .

15.4 Implementing Dinitz' Algorithm

One caveat to the implementation of Dinitz algorithm is that in several places, we need to have a comparison value which is always going to be bigger than present in the graph. For instance, the initial distance to a vertex $v \neq s$ in the modified Dijkstra's code should be ∞ , but Python has no built-in mathematical understanding of ∞ ! In those instances, we will define a variable `infty` which is larger than the largest value which we should ever see in that context.

```
def dinitz(cap_dic, source, target):

    #Our helper functions:
    # This removes 0-weight edges and cleans entries for vertices with no
    # out-edges from the dictionary.
    clean = lambda weighted_graph: {u:{v:w for v,w in nbrs.items() if w>0}
                                     for u, nbrs in weighted_graph.items()
                                     if nbrs != {} and sum(nbrs.values())>0}

    # This is the modified Dijkstra's algorithm
    def all_shortest_paths(weight_dic, source, vertex_set):
        # The longest path on n vertices has n-1 edges, so properly bigger than
        # the longest path is
        infty = len(vertex_set)+1
        dist = {v:(infty if v!=source else 0) for v in V}
        pred = {v:{}} for v in V}
        visited = []
        max_dist = 0
        is_pred = set([])
        while len(visited)<len(V):
            temp = sorted((d,v) for v,d in dist.items() if v not in visited)
            if temp == [] or temp[0][0]==infty:
                break
            else:
                u = temp[0][1]
                visited += [u]
                try:
                    for v,w in weight_dic[u].items():
                        if w>0 and dist[v] >= dist[u]+1:
                            dist[v] = dist[u]+1
                            if dist[v]>max_dist:
                                max_dist = dist[v]
                            pred[v][u]=w
                            is_pred.add(u)
                except KeyError:
                    continue
        return ({v:d for v,d in dist.items() if d<infty},
                {v:p for v,p in pred.items() if v in dist.keys()
                 and dist[v]<infty})

    # The pathfinding algorithm, with error handling which takes care of the
    # special terminate cases.
    def path_find(layers):
        # 1 more than the highest capacity in the layered graph is effectively
        # infinite
        infty = 1+max([w for d, layer in layers.items()
                       for u,nbrs in layer.items()])
```

```

        for v,w in nbrs.items())
    if any(layer=={} for layer in layers.values()):
        raise RuntimeError('No augmenting path! Empty layer in graph.')
    try:
        path = [ list(layers[0].keys())[0] ]
        flow_amount = inf
        for d in sorted(layers.keys())[:-1]:
            u = path[-1]
            v = list(layers[d][u].keys())[0]
            flow_amount = min(flow_amount, layers[d][u][v])
            path += [v]

        return (flow_amount, path)
    except KeyError:
        raise RuntimeError('No augmenting path remains in layered graph.')

def clean_to_source(layers, u, d):
    if layers[d][u] == {}:
        if d>0:
            for w,w_nbrs in layers[d-1].items():
                if u in w_nbrs.keys():
                    layers[d-1][w].pop(u)
                    if len(w_nbrs)==0:
                        layers = clean_to_source(layers, w, d-1)
            layers[d].pop(u)
    return layers

def clean_to_target(layers, v, d):
    maxd = max(layers.keys())
    if not any(v in layers[d-1][w].keys() for w in layers[d-1].keys()):
        for w in layers[d][v].keys():
            layers[d][v].pop(w)
            if d<maxd:
                layers = clean_to_target(layers, w, d+1)
    layers[d].pop(v)
    return layers

def maintain_layers(layers, source, target, weight, path):
    for d,(u,v) in enumerate(zip(path[:-1],path[1:])):
        try:
            layers[d][u][v] -= weight
            if layers[d][u][v] == 0:
                zero = layers[d][u].pop(v)
                layers = clean_to_source(layers, u, d)
                layers = clean_to_target(layers,v,d+1)
        except KeyError:
            continue
    return layers

# end of helper functions
# This is the main routine of the dinitz function

V = set(v for u in cap_dic.keys() for v in [u]+list(cap_dic[u].keys()))
# Minor error handling
if not (source in V and target in V):
    raise ValueError('Invalid choice of source and/or target vertices.')

#initialize flow

```

```
flow = {u:{v:0 for v in V if v!=u} for u in V}
a
# This is the main loop.
while True:
    # res_cap is the capacity dictionary for Delta_f
    res_cap = {u:{v:0 for v in V if v!=u} for u in V}
    for u, nbrs in cap_dic.items():
        for v, c in nbrs.items():
            res_cap[u][v] += c-flow[u][v]
            res_cap[v][u] += flow[u][v]
    res_cap = clean(res_cap)

    # modified Dijkstra first
    dist, pred = all_shortest_paths(res_cap, source, V)
    # modified Dijkstra second
    dist, succ = all_shortest_paths(pred, target, V)
    # maxd is the length of a shortest path in Lambda
    maxd = max(d for d in dist.values() if d<len(dist.keys()))

    if target not in [v for u,nbrs in succ.items() for v in nbrs.keys()]:
        # if target is not reachable via Dijkstra's algorithm, we terminate
        # the main loop.
        break

    layers = {maxd-d:{u:n for u,n in succ.items() if dist[u]==d}
              for d in dist.values() if d <= maxd}

    # iterate through finding augmenting flows in the layered structure
    try:
        while not all(val=={} for val in layers.values()):
            # use path_find to find an augmenting path; this may generate
            # a RuntimeError.
            flow_val, path = path_find(layers)

            # add the flow amount to every edge in the path.
            for i,v in enumerate(path[:-1]):
                flow[v][path[i+1]] += flow_val

            # prune the layered data structure; this is equivalent to
            # recalculating the residual capacity but uses the calculated
            # union of shortest paths, until no path from source to target
            # remains. Edges of residual capacity 0 are removed, as are
            # vertices with no outflow or no inflow.

            layers = maintain_layers(layers, source, target, flow_val, path)
    except RuntimeError:
        # If the RuntimeError was generated, then no more augmenting flows
        # exist. Go to the next iteration of the main loop.
        continue

    # Once the code reaches this point, we've broken out of the main loop, so no
    # more augmenting paths are possible.
    # The sum of flows out of the source is the value of the flow.
    flow_val = sum(f for f in flow[source].values())

    # return the value of the flow and the dictionary of edges with positive flow.
    return flow_val, clean(flow)
```