

Computational Modeling and Complexity Science

Version 0.0.10

Computational Modeling and Complexity Science

Version 0.0.10

Allen Downey

Green Tea Press

Needham, Massachusetts

Copyright © 2008 Allen Downey.

Printing history:

Fall 2008: First edition.

Green Tea Press
9 Washburn Ave
Needham MA 02492

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and with no Back-Cover Texts.

The GNU Free Documentation License is available from www.gnu.org or by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

The original form of this book is \LaTeX source code. Compiling this \LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The \LaTeX source for this book is available from <http://greenteapress.com/compmo>

Preface

This book is about data structures and algorithms, intermediate programming in Python, complexity science and the philosophy of science:

Data structures and algorithms: A data structure is a collection that contains data elements organized in a way that supports particular operations. For example, a dictionary organizes key-value pairs in a way that provides fast mapping from keys to values, but mapping from values to keys is generally slower.

An algorithm is an mechanical process for performing a computation. Designing efficient programs often involves the co-evolution of data structures and the algorithms that use them. For example, the first few chapters are about graphs, a data structure (nested dictionaries) that is a good implementation of a graph, and several graph algorithms that use this data structure.

Python programming: This book picks up where *Think Python* leaves off. I assume that you have read that book or have equivalent knowledge of Python. As always, I will try to emphasize fundamental ideas that apply to programming in many languages, but along the way you will learn some useful features that are specific to Python.

Computational modeling: A model is a simplified description of a system that is useful for simulation or analysis. Computational models are designed to take advantage of cheap, fast computation.

Philosophy of science: The models and results I will present raise a number of questions relevant to the philosophy of science, including the nature of scientific laws, theory choice, realism and instrumentalism, holism and reductionism, and Bayesian epistemology.

There are two kinds of computational models:

Continuous: Many computational models compute discrete approximations of equations that are continuous in space and time. For example, to compute the trajectory of a planet, you could describe planetary motion using differential equations and then compute a numerical approximation of the position of the planet at discrete points in time.

The fields of numerical methods and scientific computing tend to focus on these kinds of models.

Discrete: Discrete models include graphs, cellular automata, and agent-based models. They are often characterized by structure, rules and transitions rather than by equations. They tend to

be more abstract than continuous models; in some cases there is no direct correspondence between the model and a physical system.

Complexity science is an interdisciplinary field—at the intersection of mathematics, computer science and physics—that focuses on these kinds of models.

And that’s what this book is about.

Allen B. Downey
Needham MA

Allen Downey is an Associate Professor of Computer Science at the Franklin W. Olin College of Engineering.

Acknowledgements

Contributor List

If you have a suggestion or correction, please send email to compmod@greenteapress.com. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- Richard Hollands pointed out several typos.

Contents

Preface	v
1 Graphs	1
1.1 What's a graph?	1
1.2 Representing graphs	3
1.3 Random graphs	7
1.4 Connected graphs	7
1.5 Paul Erdős: peripatetic mathematician, speed freak	7
1.6 Iterators	8
1.7 Generators	9
2 Analysis of algorithms	11
2.1 Order of growth	12
2.2 Analysis of basic operations	14
2.3 Analysis of search algorithms	15
2.4 Hashtables	16
2.5 Summing lists	19
2.6 List comprehensions	21
3 Small world graphs	23
3.1 Analysis of graph algorithms	23
3.2 FIFO implementation	24
3.3 Stanley Milgram	25
3.4 Watts and Strogatz	25
3.5 Dijkstra	27
3.6 What kind of explanation is <i>that</i> ?	28

4	Scale-free networks	31
4.1	Zipf's Law	31
4.2	Cumulative distributions	32
4.3	Closed-form distributions	34
4.4	Pareto distributions	35
4.5	Barabási and Albert	37
4.6	Zipf, Pareto and power laws	38
5	Cellular Automata	43
5.1	Wolfram	43
5.2	Implementing CAs	45
5.3	Classifying CAs	47
5.4	Randomness	48
5.5	Determinism	49
5.6	Structures	50
5.7	Universality	52
5.8	Falsifiability	53
5.9	What is this a model of?	54
6	Games of Life	57
6.1	Abstract classes	57
6.2	Conway	58
6.3	Life patterns	59
6.4	Conway's conjecture	60
6.5	Realism	60
6.6	Instrumentalism	61
6.7	Turmites	62
6.8	Percolation	62

7	Self-organized criticality	65
7.1	Bak, Tang and Wiesenfeld	65
7.2	Spectral density	66
7.3	Fast Fourier Transform	68
7.4	Pink noise	69
7.5	Forest fire models	69
7.6	Reductionism and Holism	70
7.7	SOC, causation and prediction	72
8	Agent-based models	75
8.1	Schelling	75
8.2	Agent-based models	76
8.3	Traffic jams	76
8.4	Boids	78
8.5	Ants	80
8.6	Sugarscape	80
8.7	Emergence	80
9	Stochastic modeling	81
9.1	Monty Hall	81
9.2	Poincaré	83
9.3	Streaks and hot spots	83
9.4	Bayes' theorem	85

Chapter 1

Graphs

1.1 What's a graph?

To most people a graph is a visual representation of a data set, like a bar chart or an EKG. That's not what this chapter is about.

In this chapter, a **graph** is an abstraction used to model a system that contains discrete, interconnected elements. The elements are represented by **nodes** (also called **vertices**) and the interconnections are represented by **edges**.

For example, you could represent a road map with one node for each city and one edge for each road between cities. Or you could represent a social network using one node for each person, with an edge between two people if they are “friends” and no edge otherwise.

In some graphs, edges have different lengths (sometimes called “weights” or “costs”). For example, in a road map, the length of an edge might represent the distance between two cities, or the travel time, or bus fare, and so on. In a social network there might be different kinds of edges to represent different kinds of relationships: friends, business associates, etc.

Edges may be **undirected**, if they represent a relationship that is symmetric, or **directed**. In a social network, friendship is usually symmetric: if *A* is friends with *B* then *B* is friends with *A*. So you would probably represent friendship with an undirected edge. In a road map, you would probably represent a one-way street with a directed edge.

Graphs have many interesting mathematical properties, and there is a branch of mathematics called **graph theory** that studies them.

Graphs are also useful, because there are many real world problems that can be solved using **graph algorithms**. For example, Dijkstra's shortest path algorithm is an efficient way to find the shortest path from a node to all other nodes in a graph. A **path** is a sequence of nodes with an edge between each consecutive pair.

Sometimes the connection between a real world problem and a graph algorithm is obvious. In the road map example, it is not hard to imagine using a shortest path algorithm to find the route between two cities that minimizes distance (or time, or cost).

In other cases it takes more effort to represent a problem in a form that can be solved with a graph algorithm, and then interpret the solution.

For example, a complex system of radioactive decay¹ can be represented by a graph with one node for each nuclide (type of atom) and an edge between two nuclides if one can decay into the other. A path in this graph represents a decay chain.

The rate of decay between two nuclides is characterized by a decay constant, λ , measured in becquerels (Bq) or decay events per second².

In our best current model of physics, nuclear decay is a fundamentally random process, so it is impossible to predict when an atom will decay. However, given λ , the probability that an atom will decay during a short time interval dt is λdt .

In a graph with multiple decay chains, the probability of a given path is the product of the probabilities of each decay process in the path.

Now suppose you want to find the decay chain with the highest probability. You could do it by assigning each edge a “length” of $-\log \lambda$ and using a shortest path algorithm. Why? Because the shortest path algorithm adds up the lengths of the edges, and adding up log-probabilities is the same as multiplying probabilities. Also, because the logarithms are negated, the smallest sum corresponds to the largest probability. So the shortest path corresponds to the most likely decay chain.

This is an example of a common and useful process in applying graph algorithms:

Reduce a real-world problem to an instance of a graph problem.

Apply a graph algorithm to compute the result efficiently.

Interpret the result of the computation in terms of a solution to the original problem.

We will see other examples of this process soon.

Exercise 1.1 Read the Wikipedia page about graphs at [http://en.wikipedia.org/wiki/Graph_\(mathematics\)](http://en.wikipedia.org/wiki/Graph_(mathematics)) and answer the following questions:

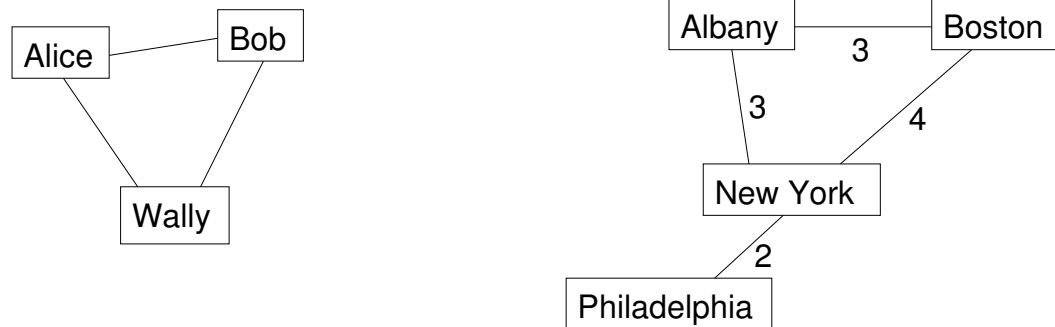
1. What is a simple graph? In the rest of this section, we will be assuming that all graphs are simple graphs. This is a common assumption for many graph algorithms—so common it is often unstated.
2. What is a regular graph? What is a complete graph? Prove that a complete graph is regular.
3. What is a path? What is a cycle?
4. What is a forest? What is a tree? Note: a graph is **connected** if there is a path from every node to every other node.

¹In developing this example I relied on [wikipedia.org/wiki/Radioactive_decay](http://en.wikipedia.org/wiki/Radioactive_decay).

²You might be more familiar with half-life, $t_{1/2}$, which is the expected time until half of a sample decays. You can convert from one characterization to the other using the relation $t_{1/2} = \ln 2 / \lambda$.

1.2 Representing graphs

Graphs are usually drawn with squares or circles for nodes and lines for edges. In the example below, the graph on the left represents a social network with three people.



In the graph on the right, the weights of the edges are the approximate travel times, in hours, between cities in the northeast United States. In this case the placement of the nodes corresponds roughly to the geography of the cities, but in general the layout of a graph is arbitrary.

To implement graph algorithms, you have to figure out how to represent a graph in the form of a data structure. But to choose the best data structure, you have to know which operations the graph should support.

To get out of this chicken-and-egg problem, I am going to present a data structure that is a good choice for many graph algorithms. Later we will come back and evaluate its pros and cons.

Here is an implementation of a graph as a dictionary of dictionaries:

```

class Graph(dict):
    def __init__(self, vs=[], es=[]):
        """create a new graph.  (vs) is a list of vertices;
        (es) is a list of edges."""
        for v in vs:
            self.add_vertex(v)

        for e in es:
            self.add_edge(e)

    def add_vertex(self, v):
        """add (v) to the graph"""
        self[v] = {}

    def add_edge(self, e):
        """add (e) to the graph by adding an entry in both directions.

        If there is already an edge connecting these Vertices, the
        new edge replaces it.
        """
        v, w = e

```

```

    self[v][w] = e
    self[w][v] = e

```

The first line declares that `Graph` inherits from the built-in type `dict`, so a `Graph` object has all the methods and operators of a dictionary.

More specifically, a `Graph` is a dictionary that maps from a `Vertex` v to an inner dictionary that maps from a `Vertex` w to an `Edge` that connects v and w . So if g is a graph, $g[v][w]$ maps to an `Edge` if there is one and raises a `KeyError` otherwise.

The `__init__` method takes a list of vertices and a list of edges as optional parameters. If they are provided, it calls `add_vertex` and `add_edge` to add the vertices and edges to the graph.

Adding a vertex to a graph means making an entry for it in the outer dictionary. Adding an edge makes two entries, both pointing to the same `Edge`. So this implementation represents an undirected graph (see Exercise 1.5).

Here is the definitions for `Vertex`:

```

class Vertex(object):
    def __init__(self, label=''):
        self.label = label

    def __repr__(self):
        return 'Vertex(%s)' % repr(self.label)

    __str__ = __repr__

```

A `Vertex` is just an object that has a `label` attribute. We will add more attributes later, as needed.

`__repr__` is a special function that returns a string representation of an object. It is similar to `__str__` except that the return value from `__str__` is intended to be readable for people, and the return value from `__repr__` is supposed to be a legal Python expression.

The built-in function `str` invokes the `__str__` method on an object; similarly the built-in function `repr` invokes `__repr__`.

In this case `Vertex.__str__` and `Vertex.__repr__` refer to the same function, so we get the same string either way.

Here is the definition for `Edge`:

```

class Edge(tuple):
    def __new__(cls, *vs):
        return tuple.__new__(cls, vs)

    def __repr__(self):
        return 'Edge(%s, %s)' % (repr(self[0]), repr(self[1]))

    __str__ = __repr__

```

`Edge` inherits from the built-in type `tuple` and overrides the `__new__` method. When you invoke an object constructor, Python invokes `__new__` to create the object and then `__init__` to initialize the attributes.

For mutable objects it is most common to override `__init__` and use the default implementation of `__new__`, but because Edges inherit from `tuple`, they are immutable, which means that you can't modify the elements of the tuple in `__init__`.

By overriding `__new__`, we can use the `*` operator to gather the parameters and use them to initialize the elements of the tuple. A precondition of this method is that there should be exactly two arguments. A more careful implementation would check.

Here is an example that creates two vertices and an edge:

```
v = Vertex('v')
w = Vertex('w')
e = Edge(v, w)
print e
```

Inside `Edge.__str__` the term `self[0]` refers to `v` and `self[1]` refers to `w`. So the output when you print `e` is:

```
Edge(Vertex('v'), Vertex('w'))
```

Now we can assemble the edge and vertices into a graph:

```
g = Graph([v,w], [e])
print g
```

The output looks like this (with a little formatting):

```
{
  Vertex('w'): {Vertex('v'): Edge(Vertex('v'), Vertex('w'))},
  Vertex('v'): {Vertex('w'): Edge(Vertex('v'), Vertex('w'))}
}
```

If you evaluate this expression you get a dictionary rather than a `Graph`, so this might not be the best representation of a `Graph`. But it turns out to be non-trivial to generate a string representation that can recreate the graph.

Exercise 1.2 In this exercise you will write `Graph` methods that will be useful for many of the `Graph` algorithms that are coming up.

Download greenteapress.com/compmo/GraphCode.py, which contains the code in this chapter. Run it as a script and make sure the test code in `main` does what you expect.

Make a copy of `GraphCode.py` called `Graph.py`. Add the following methods to `Graph`, adding test code as you go:

1. Write a method named `get_edge` that takes two vertices and returns the edge between them if it exists and `None` otherwise. Hint: use a `try` statement.
2. Write a method named `remove_edge` that takes an edge and removes all references to it from the graph.
3. Write a method named `vertices` that returns a list of the vertices in a graph.
4. Write a method named `edges` that returns a list of edges in a graph. Note that in our representation of an undirected graph there are two references to each edge.
5. Write a method named `out_vertices` that takes a `Vertex` and returns a list of the adjacent vertices (the ones connected to the given node by an edge).

6. Write a method named `out_edges` that takes a `Vertex` and returns a list of edges connected to the given `Vertex`.
7. Write a method named `add_all_edges` that starts with an edgeless `Graph` and makes a complete graph by adding edges between all pairs of vertices.

Test your methods by writing test code and checking the output. Then download `greenteapress.com/compmo/GraphWorld.py`. `GraphWorld` is a simple tool for generating visual representations of graphs. It is based on the `World` class in `Swampy`, so you might have to install `Swampy` first: you can download it from `thinkpython.com/swampy`.

Read through `GraphWorld.py` to get a sense of how it works. Then run it. It should import your `Graph.py` and then display a complete graph with 10 vertices.

Exercise 1.3 Write a method named `add_regular_edges` that starts with an edgeless graph and adds edges so that every vertex has the same degree (which the method takes as a parameter). The **degree** of a node is the number of edges it is connected to.

To create a regular graph with degree 2 you would do something like this:

```
vertices = [ ... list of Vertices ... ]
g = Graph(vertices, [])
g.add_regular_edges(2)
```

Note that it is not always possible to create a regular graph with a given degree. You should figure out and document the preconditions for this method.

To test your code, you might want to create a file named `TestGraph.py` that imports `Graph.py` and `GraphWorld.py`, then generates and displays the graphs you want to test.

Exercise 1.4 Write a `__repr__` method for `Graphs` that returns a string representation of the graph that can be evaluated to construct a new graph with the same structure and the same vertex labels.

Note: to make this work, you will probably want to override `Vertex.__new__` so that if you try to create a `Vertex` with a label that already exists, it returns a reference to the existing `Vertex` instead of creating a new one. This is a variation of the Singleton pattern which you can read about at http://en.wikipedia.org/wiki/Singleton_pattern.

Exercise 1.5 Create a file named `Digraph.py` and write a definition for a class called `Digraph` that inherits from `Graph`.

1. Look over the methods in `Graph` and decide which ones need to be overridden to implement a directed graph. Implement and test these methods.
2. Write a method named `in_edges` that takes a `Vertex v` and returns a list of the directed edges coming into `v`. Which is more efficient, `in_edges` or `out_edges`?
3. Write a method named `in_vertices` that takes a `Vertex v` and returns a list of the `Vertices w` connected by an edge from `w` to `v`.

You can download my solution to these exercises from `greenteapress.com/compmo/Graph.py` and `greenteapress.com/compmo/Digraph.py`

1.3 Random graphs

A random graph is just what it sounds like: a graph with edges generated at random. Of course, there are many random processes that can generate graphs, so there are many kinds of random graph. One of the more popular kinds is the Erdős-Rényi model, denoted $G(n, p)$, which generates graphs with n nodes, where the probability is p that there is an edge between any two nodes.

wikipedia.org/wiki/Random_graph

Exercise 1.6 Create a file named `RandomGraph.py` and define a class named `RandomGraph` that inherits from `Graph` and provides a method named `add_random_edges` that takes a probability p as a parameter and, starting with an edgeless graph, adds edges at random so that the probability is p that there is an edge between any two nodes.

1.4 Connected graphs

A graph is **connected** if there is a path from every node to every other node (see [wikipedia.org/wiki/Connectivity_\(graph_theory\)](http://wikipedia.org/wiki/Connectivity_(graph_theory))).

There is a simple algorithm to check whether a graph is connected. Start at any vertex and conduct a search (usually a breadth-first-search or BFS), marking all the vertices you can reach. Then check to see whether all vertices are marked.

You can read about breadth-first-search at wikipedia.org/wiki/Breadth-first_search. Searches are based on an operation called **visiting**: to “visit” a node, you mark it so you can tell later that it has been visited, then visit any unmarked vertices it is connected to.

A breadth-first-search visits nodes in the order they are discovered. It uses a queue or a “worklist” to keep them in order. Here’s how it works:

1. Start with any vertex and add it to the queue.
2. Remove a vertex from the queue and mark it. If it is connected to any unmarked vertices, add them to the queue.
3. If the queue is not empty, go back to Step 2.

Exercise 1.7 Write a `Graph` method named `is_connected` that returns `True` if the `Graph` is connected and `False` otherwise.

1.5 Paul Erdős: peripatetic mathematician, speed freak

Paul Erdős was a Hungarian mathematician who spent most of his career (from 1934 until his death in 1992) living out of a suitcase, visiting colleagues at universities all over the world, and authoring papers with more than 500 collaborators.

He was a notorious caffeine addict and, for the last 20 years of his life, an enthusiastic user of amphetamines. He attributed at least some of his productivity to the use of these drugs; after giving them up for a month to win a bet, he complained that mathematics had been set back by a month³.

³Much of this biography follows wikipedia.org/wiki/Paul_Erdős

In the 1960s he and Afréd Rényi wrote a series of papers introducing the Erdős-Rényi model of random graphs and studying their properties.

One of their most surprising results is the existence of abrupt changes in the characteristics of random graphs as random edges are added. They showed that for a number of graph properties there is a threshold value of the probability p below which the property is rare and above which is almost certain. This transition is sometimes called a “phase change” by analogy with physical systems that change state at some critical value of temperature (see wikipedia.org/wiki/Phase_transition).

Exercise 1.8 One of the properties that displays this kind of transition is connectedness. For a given size n , there is a critical value of p^* such that a random graph $G(n, p)$ is unlikely to be connected if $p < p^*$ and very likely to be connected if $p > p^*$.

Write a program that tests this result by generating a large number of random graphs for given values of n and p . and computes the fraction of them that are connected.

How does the abruptness of the transition depend on n ?

You can download my solution from greenteapress.com/compmo/RandomGraph.py.

1.6 Iterators

If you have read the documentation of Python dictionaries, you might have noticed the methods `iterkeys`, `itervalues` and `iteritems`. These methods are similar to `keys`, `values` and `items`, except that instead of building a new list, they return iterators.

An **iterator** is an object that provides a method named `next` that returns the next element in a sequence. Here is an example that creates a dictionary and uses `iterkeys` to traverse the keys.

```
>>> d = dict(a=1, b=2)
>>> iter = d.iterkeys()
>>> print iter.next()
a
>>> print iter.next()
b
>>> print iter.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The first time `next` is invoked, it returns the first key from the dictionary (the order of the keys is arbitrary). The second time it is invoked, it returns the second element. The third time, and every time thereafter, it raises a `StopIteration` exception.

An iterator can be used in a `for` loop; for example, the following is a common idiom for traversing the key-value pairs in a dictionary:

```
for k, v in d.iteritems():
    print k, v
```

In this context, `iteritems` is likely to be faster than `items` because it doesn't have to build the entire list of tuples; it reads them from the dictionary as it goes along.

But it is only safe to use the iterator methods if you do not add or remove dictionary keys inside the loop. Otherwise you get an exception:

```
>>> d = dict(a=1)
>>> for k in d.iterkeys():
...     d['b'] = 2
...
RuntimeError: dictionary changed size during iteration
```

Another limitation of iterators is that they do not support index operations.

```
>>> iter = d.iterkeys()
>>> print iter[1]
TypeError: 'dictionary-keyiterator' object is unsubscriptable
```

If you need indexed access, you should use `keys`. Alternatively, the Python module `itertools` provides many useful iterator functions.

A user-defined object can be used as an iterator if it provides methods named `next` and `__iter__`. The following example is an iterator that always returns `True`:

```
class AllTrue(object):
    def next(self):
        return True

    def __iter__(self):
        return self
```

The `__iter__` method for iterators returns the iterator itself. This protocol makes it possible to use iterators and sequences interchangeably in many contexts.

Iterators like `AllTrue` can represent an infinite sequence. They are useful as an argument to `zip`:

```
>>> print zip('abc', AllTrue())
[('a', True), ('b', True), ('c', True)]
```

1.7 Generators

For many purposes the easiest way to make an iterator is to write a **generator**, which is a function that contains a `yield` statement. `yield` is similar to `return`, except that the state of the running function is stored and can be resumed.

For example, here is a generator that yields successive letters of the alphabet:

```
def generate_letters():
    for c in 'abc':
        yield c
```

If you call this function, the return value is an iterator:

```
>>> iter = generate_letters()
>>> print iter
<generator object at 0xb7d4ce4c>
>>> print iter.next()
a
>>> print iter.next()
b
```

And you can use an iterator in a for loop:

```
>>> for letter in generate_letters():
...     print letter
...
a
b
c
```

A generator with an infinite loop returns an iterator that never terminates. For example, here's a generator that cycles through the letters of the alphabet:

```
def alphabet_cycle():
    while 1:
        for c in string.lowercase:
            yield c
```

Exercise 1.9 Write a generator that yields an infinite sequence of alpha-numeric identifiers, starting with a1 through z1, then a2 through z2, and so on.

Chapter 2

Analysis of algorithms

Analysis of algorithms is the branch of computer science that studies the performance of algorithms, especially their run time and space requirements.

The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide program design decisions.

During the 2008 United States Presidential Campaign, candidate Barack Obama was asked to perform an impromptu analysis when he visited Google. Chief executive Eric Schmidt jokingly asked him for “the most efficient way to sort a million 32-bit integers.” Obama had apparently been tipped off, because he quickly replied, “I think the bubble sort would be the wrong way to go.”

This is true: bubble sort is conceptually simple but slow for large datasets. The answer Schmidt was probably looking for is “radix sort” (see wikipedia.org/wiki/Radix_sort)¹.

So the goal of algorithmic analysis is to make meaningful comparisons between algorithms, but there are some problems:

- The relative performance of the algorithms might depend on characteristics of the hardware, so one algorithm might be faster on Machine A, another on Machine B. The general solution to this problem is to specify a **machine model** and analyze the number of steps, or operations, an algorithm requires under a given model.
- Relative performance might depend on the details of the dataset. For example, some sorting algorithms run faster if the data are already partially sorted; other algorithms run slower in this case. A common way to avoid this problem is to analyze the **worst case** scenario. It is also sometimes useful to analyze average case performance, but it is usually harder, and sometimes it is not clear what set of cases to average over.

¹But if you get a question like this in an interview, I think a better answer is, “The fastest way to sort a million integers is to use whatever sort function is provided by the language I’m using. Its performance is good enough for the vast majority of applications, but if it turned out that my application was too slow, I would use a profiler to see where the time was being spent. If it looked like a faster sort algorithm would have a significant effect on performance, then I would look around for a good implementation of radix sort.”

- Relative performance also depends on the size of the problem. A sorting algorithm that does better on small lists might be much slower on long lists. The usual solution to this problem is to express run time (or number of operations) as a function of problem size, and to **compare the functions asymptotically** as the problem size increases.

The good thing about this kind of comparison is that it lends itself to simple classification of algorithms. For example, if I know that the run time of Algorithm A tends to be proportional to the size of the input, n , and Algorithm B tends to be proportional to n^2 , then it would be reasonable to expect A to be faster than B for large values of n .

This kind of analysis comes with some caveats, but we'll get to that later.

2.1 Order of growth

Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input: Algorithm A takes $100n + 1$ steps to solve a problem with size n ; Algorithm B takes $n^2 + n + 1$ steps.

The following table shows the run time of these algorithms for different problem sizes:

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

At $n = 10$, Algorithm A looks pretty bad; it takes almost 10 times longer than Algorithm B. But for $n = 100$ they are about the same, and for larger values A is much better.

The fundamental reason is that for large values of n , any function that contains an n^2 term will grow faster than a function whose leading term is n . The **leading term** is the term with the highest exponent.

For Algorithm A, the leading term has a large coefficient, 100, which is why B does better than A for small n . But regardless of the coefficients, there will always be some value of n where $an^2 > bn$.

The same argument applies to the non-leading terms. Even if the run time of Algorithm A were $n + 1000000$, it would still be better than Algorithm B for sufficiently large n .

In general, we expect an algorithm with a smaller leading term to be a better algorithm for large problems, but for smaller problems, there may be a **crossover point** where another algorithm is better. The location of the crossover point depends on the details of the algorithms, the inputs and the hardware, so it is usually ignored for purposes of algorithmic analysis, but that doesn't mean you can forget about it.

If two algorithms have the same leading order term, it is hard to say which is better; again, the answer will depend on the details. So for purposes of algorithmic analysis, functions with the same leading term are considered equivalent, even if they have different coefficients.

An **order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example, $2n$, $100n$ and $n + 1$ belong to the same order of growth, which is written $O(n)$ in “Big-Oh notation” and often called “linear” because every function in the set grows linearly with n .

All functions with the leading term n^2 belong to $O(n^2)$; they are called “quadratic,” which is a fancy word for functions with the leading term n^2 .

The following table shows some of the orders of growth that appear most commonly in algorithmic analysis, in increasing order of badness.

Order of growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	“en log en”
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any c)

For the logarithmic terms, the base of the logarithm doesn’t matter; changing bases is the equivalent of multiplying by a constant, which doesn’t change the order of growth. Similarly, all exponential functions belong to the same order of growth regardless of the base of the exponent. Exponential functions grow very quickly, so exponential algorithms are only useful for small problems.

Exercise 2.1 Read the Wikipedia page on Big-Oh notation at wikipedia.org/wiki/Big_O_notation and answer the following questions:

1. What is the order of growth of $n^3 + n^2$? What about $1000000n^3 + n^2$. What about $n^3 + 1000000n^2$?
2. What is the order of growth of $(n^2 + n) * (n + 1)$? Before you start multiplying, remember that you only need the leading term.
3. If f is in $O(g)$, for some unspecified function g , what can we say about $af + b$?
4. If f_1 and f_2 are in $O(g)$, what can we say about $f_1 + f_2$?
5. If f_1 is in $O(g)$ and f_2 is in $O(h)$, what can we say about $f_1 + f_2$?
6. If f_1 is in $O(g)$ and f_2 is $O(h)$, what can we say about $f_1 * f_2$?

Programmers who care about performance often find this kind of analysis hard to swallow. They have a point: sometimes the coefficients and the non-leading terms make a real difference. And sometimes the details of the hardware, the programming language, and the characteristics of the input make a big difference. And for small problems asymptotic behavior is irrelevant.

But if you keep those caveats in mind, algorithmic analysis is a useful tool. At least for large problems, the “better” algorithm is usually better, and sometimes it is *much* better. The difference between two algorithms with the same order of growth is usually a constant factor, but the difference between a good algorithm and a bad algorithm is infinite²!

²A n goes to infinity.

2.2 Analysis of basic operations

Most arithmetic operations are constant time; multiplication usually takes longer than addition and subtraction, and division takes even longer, but these run times don't depend on the magnitude of the operands. Very large integers are an exception; in that case the run time increases linearly with the number of digits.

Indexing operations—reading or writing elements in a sequence or dictionary—are also constant time, regardless of the size of the data structure.

A `for` loop that traverses a sequence or dictionary is usually linear, as long as all of the operations in the body of the loop are constant time. For example, adding up the elements of a list is linear:

```
total = 0
for x in t:
    total += x
```

The built-in function `sum` is also linear because it does the same thing, but it tends to be faster because it is a more efficient implementation; in the language of algorithmic analysis, it has a smaller leading coefficient.

If you use the same loop to “add” a list of strings, the run time is quadratic because string addition is linear in the length of the strings. The string method `join` is usually faster because it is linear in the total length of the strings.

As a rule of thumb, if the body of a loop is in $O(n^a)$ then the whole loop is in $O(n^{a+1})$. The exception is if you can show that the loop exits after a constant number of iterations. If a loop runs k times regardless of n , then the loop is in $O(n^a)$, even for large k . But if it runs n/k times, the loop is in $O(n^{a+1})$, even for large k .

Most string and tuple operations are linear, except indexing and `len`, which are constant time. The built-in functions `min` and `max` are linear. The run-time of a slice operation is proportional to the length of the output, but independent of the size of the input.

All string methods are linear, but if the lengths of the strings are bounded by a constant—for example, operations on single characters—they are considered constant time.

Most list methods are linear, but there are some exceptions:

- Adding an element to the end of a list is constant time on average; when it runs out of room it occasionally gets copied to a bigger location, but the total time for n operations is $O(n)$, so we say that the “amortized” time for one operation is $O(1)$.
- Removing an element from the end of a list is constant time.
- Sorting is $O(n \log n)$.

Most dictionary operations and methods are constant time, but there are some exceptions:

- The run time of `copy` is proportional to the number of elements, but not the size of the elements (it copies references, not the elements themselves). The run time of `update` is proportional to the size of the dictionary passed as a parameter, not the dictionary being updated.

- `keys`, `values` and `items` are linear because they return new lists; `iterkeys`, `itervalues` and `iteritems` are constant time because they return iterators. But if you loop through the iterators, the loop will be linear. Using the “`iter`” functions saves some overhead, but it doesn’t change the order of growth (unless the number of items you access is bounded).

The performance of dictionaries is one of the minor miracles of computer science. We will see how they work in Section 2.4.

Exercise 2.2 Read the Wikipedia page on sorting algorithms at wikipedia.org/wiki/Sorting_algorithm and answer the following questions:

1. What is a “comparison sort?” What is the best worst-case order of growth for a comparison sort? What is the best worst-case order of growth for any sort algorithm?
2. What is the order of growth of bubble sort, and why does Barack Obama think it is “the wrong way to go.”
3. What is the order of growth of radix sort?
4. What is a stable sort and why might it matter in practice?
5. What is the worst sorting algorithm (that has a name)?
6. What sort algorithm does the C library use? What sort algorithm does Python use? Are these algorithms stable? You might have to Google around to find these answers.
7. Many of the non-comparison sorts are linear, so why does Python use an $O(n \log n)$ comparison sort?

2.3 Analysis of search algorithms

A **search** is an algorithm that takes a collection and a target item and determines whether the target is in the collection, often returning the index of the target.

The simplest search algorithm is a “linear search,” which traverses the items in the collection in order, stopping if it finds the target. In the worst case it has to traverse the entire collection, so the run time is linear, hence the name.

The `in` operator for sequences uses a linear search; so do string methods like `find` and `count`.

If the elements of the sequence are in order, you can use a **bisection search**, which is $O(\log n)$. Bisection search is similar to the algorithm you probably use to look a word up in a dictionary. Instead of starting at the beginning and checking each item in order, you start with the item in the middle and check whether the word you are looking for comes before or after. If it comes before, then you search the first half of the sequence the same way. Otherwise you search the second half. Either way, you cut the number of remaining items in half.

If the sequence has 1,000,000 items, it will take about 20 steps to find the word or conclude that it’s not there. So that’s about 50,000 times faster than a linear search.

Exercise 2.3 Write a function called `bisect` that takes a sorted list and a target value and returns the index of the value in the list, if it's there, or `None` if it's not.

Or you could read the documentation of the `bisect` module and use that!

Bisection search can be much faster than linear search, but it requires the sequence to be in order, which might require extra work.

There is another data structure, called a **hashtable** that is even faster—it can do a search in constant time—and it doesn't require the items to be sorted. Python dictionaries are implemented using hashtables, which is why most dictionary operations, including the `in` operator, are constant time.

2.4 Hashtables

To explain how hashtables work and why their performance is so good, I will start with a simple implementation of a map and gradually improve it until it's a hashtable.

I will use Python to demonstrate these implementations, but in real life you wouldn't write code like this in Python; you would just use a dictionary! So for the rest of this chapter, you have to imagine that dictionaries don't exist and you want to implement a data structure that maps from keys to values. The basic operations you have to implement are:

`add(k, v)`: Add a new item that maps from key `k` to value `v`. With a Python dictionary, `d`, this operation is written `d[k] = v`.

`get(target)`: Look up and return the value that corresponds to key `target`. With a Python dictionary, `d`, this operation is written `d[target]` or `d.get(target)`.

For now, I will assume that each key only appears once. The simplest implementation of this interface uses a list of tuples, where each tuple is a key-value pair.

```
class LinearMap(list):

    def add(self, k, v):
        self.append((k, v))

    def get(self, target):
        for key, val in self:
            if key == target:
                return val
        raise KeyError
```

`add` uses `list.append`, which is constant time. `get` uses a `for` loop to search the list: if it finds the target key it returns the corresponding value; otherwise it raises a `KeyError`. So `get` is linear.

An alternative is to keep the list sorted by key. Then `get` could use a bisection search, which is $O(\log n)$. But inserting a new item in the middle of a list is linear, so this might not be the best option. There are other data structures³ that can implement `add` and `get` in log time, but that's still not as good as a hashtable, so let's move on.

³See wikipedia.org/wiki/Red-black_tree.

One way to improve `LinearMap` is to break the list of key-value pairs into smaller lists. Here's an implementation called `BetterMap`, which is a list of 100 `LinearMaps`. As we'll see in a second, the order of growth for `get` is still linear, but `BetterMap` is a step on the path toward hashtables:

```
class BetterMap(list):

    def __init__(self, n=100):
        self.add_maps(n)

    def add_maps(self, n):
        for i in range(n):
            self.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self)
        return self[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

The `init` method uses `add_maps` to initialize itself with 100 `LinearMaps`.

When `add` and `get` are invoked, they use `find_map` to figure out which map to put the new item into, or which map to search.

`find_map` uses the built-in function `hash`, which takes almost any Python object and returns an integer. A limitation of this implementation is that it only works with hashable keys. Mutable types like lists and dictionaries are unhashable.

Hashable objects that are considered equal will return the same hash value, but the converse is not necessarily true: two different objects can return the same hash value.

`find_map` uses the modulus operator to wrap the hash values into the range from 0 to `len(self)`, so the result is a legal index into the `BetterMap`. Of course, this means that many different hash values will wrap into the same index. But if the hash function spreads things out pretty evenly (which is what hash functions are designed to do), then we expect $n/100$ items per `LinearMap` on average.

Since the run time of `LinearMap.get` is proportional to the number of items, we expect `BetterMap` to be about 100 times faster than `LinearMap`. The order of growth is still linear, but the leading coefficient is smaller. That's nice, but still not as good as a hashtable.

Here (finally) is the crucial idea that makes hashtables fast: if you can keep the maximum length of the `LinearMaps` bounded by a constant, then `LinearMap.get` is constant time. All you have to do is keep track of the number items and when the maximum (or average) number of items per `LinearMap` exceeds a threshold, resize the hashtable by adding more `LinearMaps`.

Here is an implementation of a hashtable:

```

class HashMap(BetterMap):

    def __init__(self):
        BetterMap.__init__(self, 2)
        self.num = 0

    def add(self, k, v):
        if self.num == len(self):
            self.resize()

        BetterMap.add(self, k, v)
        self.num += 1

    def resize(self):
        pairs = []
        for t in self:
            pairs.extend(t)

        self.add_maps(len(self))
        for k, v in pairs:
            BetterMap.add(self, k, v)

```

HashMap inherits from BetterMap; it overrides `__init__` and `add` and defines a new method named `resize`.

`__init__` starts with just 2 LinearMaps and initializes `num`, which keeps track of the number of items.

`add` checks the number of items and the number of LinearMaps; if they are equal, then the average number of items per LinearMap is 1, so it resizes the HashMap.

`resize` uses `list.extend` to make a list of the items in the HashMap. Then it uses `BetterMap.add_map` to double the number of LinearMaps. Finally, it “rehashes” the items that were already in the HashMap.

Rehashing is necessary because changing the number of LinearMaps changes the denominator of the modulus operator in `find_map`. That means that some objects that used to wrap into the same LinearMap will get split up (which is what we wanted, right?).

Making the list of items, adding more LinearMaps, and rehashing are all linear, so `resize` is linear, which might seem bad, since I promised that `add` would be constant time. But remember that we don’t have to resize every time, so `add` is usually constant time and only occasionally linear. The total amount of work to run `add` n times is proportional to n , so the average time of each `add` is constant time!

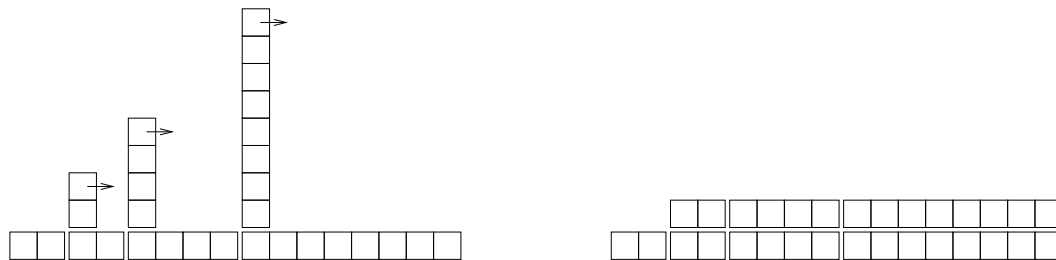
To see how this works, think about starting with an empty HashTable and adding a sequence of items. We start with 2 LinearMaps, so the first 2 adds are fast (no resizing required). Let’s say that they take one unit of work each. The next add requires a resize, so we have to rehash the first two items (let’s call that 2 more units of work) and then add the third item (one more unit). Adding the next item costs 1 unit, so the total so far is 6 units of work for 4 items.

The next add costs 5 units, but the next three are only one unit each, so the total is 14 units for the first 8 adds.

The next add costs 9 units, but then we can add 7 more before the next resize, so the total is 30 units for the first 16 adds.

After 32 adds, the total cost is 62 units, and I hope you are starting to see a pattern. After n adds, where n is a power of two, the total cost is $2n - 2$ units, so the average work per add is a little less than 2 units. When n is a power of two, that's the best case; for other values of n the average work is a little higher, but that's not important. The important thing is that it is constant time!

The following figure shows how this works graphically. Each block represents a unit of work. The columns show the total work for each add in order from left to right: the first two adds cost 1 units, the third costs 3 units, etc.



The extra work of rehashing appears as a sequence of increasingly tall towers with increasing space between them. Now if you knock over the towers, amortizing the cost of resizing over all adds, you can see graphically that the total cost after n adds is $2n - 2$.

An important feature of this algorithm is that when we resize the HashTable it grows geometrically; that is, we multiply the size by a constant. It turns out that if you increase the size arithmetically—adding a fixed number each time—the average time per add is linear.

You can download my implementation of HashMap from greenteapress.com/compmo/Map.py, but remember that there is no reason to use it; if you want a map, just use a Python dictionary.

Exercise 2.4 Write an implementation of the dictionary interface called `TreeMap` that uses a red-back tree to perform `add` and `get` in log time.

2.5 Summing lists

In my first draft of `HashMap.resize` I used the built-in function `sum` to make the list of items:

```
def resize(self):
    pairs = sum(self, [])
    self.add_maps(len(self))
    for k, v in pairs:
        BetterMap.add(self, k, v)
```

This implementation is concise, and it works, but it is what Barack Obama would call “the wrong way to go.”

The implementation of `sum` looks something like this⁴:

⁴I assume it is actually written in C, but for algorithmic analysis, language doesn't matter.

```
def badsum(t, init):
    total = init
    for x in t:
        total += x
    return total
```

The nice thing about this implementation is that it is polymorphic; it works with any type that supports the `+` operation.

The problem is that when `t` is a list of lists, this function is quadratic. The reason is that the `+` operator creates a new list each time through the loop. If you imagine adding one element per iteration, then the first iteration takes 1 unit of work, the second takes 2, the third takes 3, and so on. The total after n iterations is $n(n+1)/2$, which is in $O(n^2)$.

If `resize` is quadratic then `HashMap.add` is linear, so this implementation has a **performance error**, which means that I implemented an algorithm with an order of growth worse than the algorithm I meant to implement.

Exercise 2.5 Measure the run time of `sum` for a list of lists with a range of sizes, and confirm that it is quadratic. Then run the same experiment using `extend` and confirm that it is linear.

A simple way to measure the run time of a program is to use the function `times` in the `os` module, which returns a tuple of floats indicating the time your process has used (see the documentation for details). I use a function `etime`, which returns the sum of “user time” and “system time” which is usually what we care about for performance measurement:

```
import os

def etime():
    """See how much user and system time this process has used
    so far and return the sum."""

    user, sys, chuser, chsys, real = os.times()
    return user+sys
```

To measure the elapsed time of a function you can call `etime` twice and compute the difference:

```
start = etime()

# put the code you want to measure here

end = etime()
elapsed = end - start
```

Alternatively, if you use IPython, you can use the `timeit` command (see ipython.scipy.org).

If an algorithm is quadratic, then you expect the run time, t as a function of input size, n , to look like this:

$$t = an^2 + bn + c$$

Where a , b and c are unknown coefficients. If you take the log of both sides you get:

$$\log t \sim \log a + 2 \log n$$

For large values of n , the non-leading terms are insignificant and this approximation is pretty good. So if you plot t versus n on a log-log scale, you should see a straight line with slope 2.

Plot your results on a log-log scale and estimate the slope of the line. You might have to experiment to find values of n that are big enough to yield good measurements but not so big that the program runs forever.

Write an implementation of `sum` using `extend`. Repeat your measurements and confirm that it is linear.

You can download my solution from greenteapress.com/compmo/sumlist.py.

2.6 List comprehensions

One of the most common programming patterns is called a **map**: it involves traversing a list and performing a computation using each element of the list. The result is a new list that contains the results of the computations.

A dictionary is also sometimes called a map or a “mapping type,” but that is a different sense of the word. I’m sorry if the vocabulary is confusing, but we are stuck with it.

Here is an example that maps the square operator, `**2`, onto a list of numbers and accumulates a list that contains the square of each item:

```
res = []
for x in t:
    res.append(x**2)
```

This pattern is so common that Python provides a more concise syntax for it, called a **list comprehension**⁵

```
res = [x**2 for x in t]
```

This expression yields the same result as the previous loop. List comprehensions tend to be fast because the loop is executed in C rather than Python. The drawback is that they can be harder to debug.

List comprehensions can include an `if` clause that selects a subset of, or “filters”, the items. The following example makes a list of only the positive elements of `t`:

```
res = [x for x in t if x > 0]
```

The following is an idiom for making a list of tuples, where each tuple is a value and a key from a dictionary:

```
res = [(v,k) for k,v in d.iteritems()]
```

⁵In this context, the sense of “comprehend” is something like “contain” rather than “understand.” See wikipedia.org/wiki/List_comprehension.

In this case it is safe to use `iteritems`, rather than `items`, because the loop does not modify the dictionary; and it is likely to be faster because it doesn't have to make a list, just an iterator.

It is also possible to nest `for` loops inside a list comprehension. The following example builds a list of tuples, where each tuple is a pair of different vertices from the list `vs`:

```
res = [(v, w) for v in vs for w in vs if v is not w]
```

Now that's pretty concise!

Exercise 2.6 Can you think of a way to use a list comprehension to concatenate a list of lists into a single list, as in Exercise 2.5?

Chapter 3

Small world graphs

3.1 Analysis of graph algorithms

The order of growth for a graph algorithm is usually expressed as a function of $|V|$, the number of vertices, and $|E|$, the number of edges.

The order of growth for BFS is $O(|V| + |E|)$, which is a convenient way to say that the run time grows in proportion to either $|V|$ or $|E|$, whichever is “bigger.”

To see why, think about these four operations:

Adding a vertex to the queue: this happens once for each vertex, so the total cost is in $O(|V|)$.

Removing a vertex from the queue: this happens once for each vertex, so the total cost is in $O(|V|)$.

Marking a vertex “visited”: this happens once for each vertex, so the total cost is in $O(|V|)$.

Checking whether a vertex is marked: this happens once each edge, so the total cost is in $O(|E|)$.

Adding them up, we get $O(|V| + |E|)$. If we know the relationship between $|V|$ and $|E|$, we can simplify this expression. For example, in a regular graph, the number of edges is in $O(|V|)$ so BFS is linear in $|V|$. In a complete graph, the number of edges is in $O(|V|^2)$ so BFS is quadratic in $|V|$.

Of course, this analysis is based on the assumption that all four operations—adding and removing vertices, marking and checking marks—are constant time.

Marking vertices is easy. You can add an attribute to the `Vertex` objects or use a dictionary¹ and the `in` operator to keep track of marked vertices.

But making a first-in-first-out (FIFO) queue that can add and remove vertices in constant time turns out to be non-trivial.

¹Or a set. See docs.python.org/lib/types-set.html.

3.2 FIFO implementation

A FIFO is a data structure that provides the following operations:

append: Add a new item to the end of the queue.

pop: Remove and return the item at the front of the queue.

There are several good implementations of this data structure. One is the **doubly-linked list**, which you can read about at wikipedia.org/wiki/Doubly-linked_list. Another is a circular buffer, which you can read about at wikipedia.org/wiki/Circular_buffer.

Exercise 3.1 Write an implementation of a FIFO using either a doubly-linked list or a circular buffer.

Yet another possibility is to use a Python dictionary augmented with two indices: `nextin` keeps track of the back of the queue; `nextout` keeps track of the front. Here is an implementation²:

```
class DictQueue(dict):
    def __init__(self):
        self.nextin = 0
        self.nextout = 0

    def append(self, item):
        self.nextin += 1
        self[self.nextin] = item

    def pop(self):
        self.nextout += 1
        return dict.pop(self, self.nextout)
```

`append` increments `nextin` and then stores the new item in the dictionary; both operations are constant time³.

`pop` increments `nextout` and then uses `dict.pop` to remove and return the item at the end of the queue. Again, both operations are constant time.

As an alternative, the Python `collections` module provides an object called a deque, which stands for “double-ended queue”. It is supposed to be pronounced “deck,” but many people say “deek.” A Python deque can be adapted very easily to implement a FIFO.

You can read about deques at wikipedia.org/wiki/Deque and get the details of the Python implementation at docs.python.org/lib/deque-objects.html.

Exercise 3.2 The following implementation of a BFS⁴ contains two performance errors. What are they? What is the actual order of growth for this algorithm?

²Based on a recipe in Martelli and Ascher, *Python Cookbook*, O’Reilly 2002

³When the index exceeds the capacity of an `int`, Python switches to `long`. After that, the increment and hash operations are proportional to the number of digits in the index, which is logarithmic in the number of items that have been added.

⁴This was the implementation at wikipedia.org/wiki/Breadth-first_search before I fixed it.

```
def bfs(top_node, visit):
    """Breadth-first search on a graph, starting at top_node."""
    visited = set()
    queue = [top_node]
    while len(queue):
        curr_node = queue.pop(0)    # Dequeue
        visit(curr_node)           # Visit the node
        visited.add(curr_node)
        # Enqueue non-visited and non-enqueued children
        queue.extend(c for c in curr_node.children
                     if c not in visited and c not in queue)
```

3.3 Stanley Milgram

Stanley Milgram was an American social psychologist who conducted two of the most famous experiments in social science, the Milgram experiment, which studied people’s obedience to authority (wikipedia.org/wiki/Milgram_experiment) and the Small World Experiment wikipedia.org/wiki/Small_world_phenomenon, which studied the structure of social networks.

In the Small World Experiment, Milgram sent a package to a number of randomly-chosen people in Wichita, Kansas, with instructions asking them to forward an enclosed letter to a target person, identified by name and occupation, in Sharon, Massachusetts (which is the town near Boston where I grew up). The subjects were told that they could mail the letter directly to the target person only if they knew him personally; otherwise they were instructed to send it, and the same instructions, to a relative or friend they thought would be more likely to know the target person.

Many of the letters were never delivered, but of the ones that were it turned out that the average path length—the number of times the letters were forwarded—was about six. This result was taken to confirm previous observations (and speculations) that the typical distance between any two people in a social network is about “six degrees of separation.”

This conclusion is surprising because most people expect social networks to be localized—people tend to live near their friends—and in a graph with local connections, path lengths tend to increase in proportion to geographical distance. For example, most of my friends live nearby, so I would guess that the average distance between nodes in a social network is about 50 miles. Wichita is about 1600 miles from Boston, so if Milgram’s letters traversed typical links in the social network, they should have taken 32 hops, not six.

3.4 Watts and Strogatz

In 1998 Duncan Watts and Steven Strogatz published a paper in *Nature*, “Collective dynamics of ‘small-world’ networks,” that proposed an explanation for the small world phenomenon.

Watts and Strogatz started with two kinds of graph that were well understood: random graphs and regular graphs. They looked at two properties of these graphs, clustering and path length.

Clustering is a measure of the “cliquishness” of the graph. In a graph, a **clique** is a subset of nodes that are all connected to each other; in a social network, a clique is a set of friends

who all know each other. Watts and Strogatz defined a clustering coefficient that quantifies the likelihood that two nodes that are connected to the same node are also connected to each other.

Path length is a measure of the average distance between two nodes, which corresponds to the degrees of separation in a social network.

Their initial result was what you might expect: regular graphs have high clustering and high path lengths; random graphs with the same size tend to have low clustering and low path lengths. So neither of these is a good model of social networks, which seem to combine high clustering with short path lengths.

Their goal was to create a **generative model** of a social network. A generative model tries to explain a phenomenon by modeling a process that builds or leads to the phenomenon. In this case Watts and Strogatz proposed a process for building small-world graphs:

1. Start with a regular graph with n nodes and degree k .
2. Choose a subset of the edges in the graph and “rewire” them by replacing them with random edges.

The proportion of edges that are rewired is a parameter, p , that controls how random the graph is. With $p = 0$, the graph is regular; with $p = 1$ it is random.

Watts and Strogatz found that small values of p yield graphs with high clustering, like a regular graph, and low path lengths, like a random graph.

Exercise 3.3 Read the Watts and Strogatz paper and answer the following questions:

1. What process do Watts and Strogatz use to rewire their graphs?
2. What is the definition of the clustering coefficient $C(p)$?
3. What is the definition of the average path length $L(p)$?
4. What real-world graphs did Watts and Strogatz look at? What evidence do they present that these graphs have the same structure as the graphs generated by their model?

Exercise 3.4 Create a file named `SmallWorldGraph.py` and define a class named `SmallWorldGraph` that inherits from `RandomGraph`.

1. Write a method called `rewire` that takes a probability p as a parameter and, starting with a regular graph, rewires the graph using Watts and Strogatz’s algorithm.
2. Write a method called `clustering_coefficient` that computes and returns the clustering coefficient as defined in the paper.
3. Make a graph that replicates the line marked $C(p)/C(0)$ in Figure 2 of the paper. In other words, confirm that the clustering coefficient drops off slowly for small values of p .

Before we can replicate the other line, we have to learn about shortest path algorithms.

3.5 Dijkstra

Edsger W. Dijkstra was a Dutch computer scientist who invented an efficient shortest-path algorithm (see wikipedia.org/wiki/Dijkstra's_algorithm). He also invented the semaphore, which is a data structure used to coordinate programs that communicate with each other (see [wikipedia.org/wiki/Semaphore_\(programming\)](http://wikipedia.org/wiki/Semaphore_(programming)) and Downey, *The Little Book of Semaphores*).

Dijkstra is also famous (and notorious) as the author of a series of essays on computer science. Some, like “A Case against the GO TO Statement,” have had a profound effect on programming practice. Others, like “The Cruelty of Really Teaching Computing Science,” are entertaining in their cantankerousness, but less effective.

Dijkstra’s algorithm solves the “single source shortest path problem,” which means that it finds the minimum distance from a given “source” node to every other node in the graph (or at least every connected node).

We will start with a simplified version of the algorithm that considers all edges the same length. The more general version works with any non-negative edge lengths.

The simplified version is similar to the breadth-first search in Section 1.4 except that instead of marking visited nodes, we label them with their distance from the source. Initially all nodes are labeled with an infinite distance. Like a breadth-first search, Dijkstra’s algorithm uses a queue of discovered unvisited nodes.

1. Label the source node with distance 0 and add it to the queue.
2. Remove a vertex from the queue and call its distance d . Find the vertices it is connected to. For each connected vertex still labeled with distance infinity, replace the distance with $d + 1$ and add it to the queue.
3. If the queue is not empty, go back to Step 2.

The first time you execute Step 2, the only node in the queue has distance 0. The second time, the queue contains all nodes with distance 1. Once those nodes are processed, the queue contains all nodes with distance 2, and so on.

So when a node is discovered for the first time, it is labelled with the distance $d + 1$, which is the shortest path to that node. It is not possible that you will discover a shorter path later, because if there were a shorter path, you would have discovered it sooner. Of course, that is not a proof of the correctness of the algorithm, but it is a sketch of the structure of the proof, which is by contradiction.

In the more general case, where the edges have different lengths, it is possible to discover a shorter path after you have discovered a longer path, so a little more work is required.

Exercise 3.5 Write an implementation of Dijkstra’s algorithm and use it to compute the average path length of a SmallWorldGraph.

Make a graph that replicates the line marked $L(p)/L(0)$ in Figure 2 of the Watts and Strogatz paper. Confirm that the average path length drops off quickly for small values of p . What is the range of values for p that yield graphs with high clustering and low path lengths?

Exercise 3.6 A natural question about the Watts and Strogatz paper is whether the small world phenomenon is specific to their generative model or whether other similar models yield the same qualitative result (high clustering and low path lengths).

To answer this question, choose a variation of the Watts and Strogatz model and replicate their Figure 2. There are two kinds of variation you might consider:

- Instead of starting with a regular graph, start with another graph with high clustering. One option is a locally-connected graph where vertices are placed at random locations in the plane and each vertex is connected to its nearest k neighbors.
- Experiment with different kinds of rewiring.

If a range of similar models yield similar behavior, we would say that the results of the paper are **robust**.

Exercise 3.7 To compute the average path length in a `SmallWorldGraph`, you probably ran Dijkstra’s single-source shortest path algorithm for each node in the graph. In effect, you solved the “all-pairs shortest path” problem, which finds the shortest path between all pairs of nodes.

1. Find an algorithm for the all-pairs shortest path problem and implement it. Compare the run time with your “all-source Dijkstra” algorithm.
2. Which algorithm gives better order-of-growth run time as a function of the number of vertices and edges? Why do you think Dijkstra’s algorithm does better than the order-of-growth analysis suggests?

3.6 What kind of explanation is *that*?

If you asked me why planetary orbits are approximately elliptical, I might start by modeling a planet and a star as point masses; I would look up the law of universal gravitation at wikipedia.org/wiki/Newton's_law_of_universal_gravitation and use it to write a differential equation for the motion of the planet. Then I would either derive the orbit equation or, more likely, look it up at wikipedia.org/wiki/Orbit_equation. With a little algebra, I could derive the conditions that yield an elliptical orbit. Then I would argue that the objects we consider planets satisfy these conditions.

People, or at least scientists, are generally satisfied with this kind of explanation. One of the reasons for its appeal is that the assumptions and approximations in the model seem reasonable. Planets and stars are not really point masses, but the distances between them are so big that their actual sizes are negligible. Planets in the same solar system can affect each others’ orbits, but the effect is usually small. And we ignore relativistic effects, again on the assumption that they are small.

This explanation is also appealing because it is equation-based. We can express the orbit equation in a closed form, which means that we can compute orbits very efficiently. It also means that we can derive general expressions for the orbital velocity, orbital period, and other quantities.

Finally, I think this kind of explanation is appealing because it has the form of a mathematical proof. It starts from a set of axioms and derives the result by logic and analysis. But it is important to

remember that the proof pertains to the model and not the real world. That is, we can prove that an idealized model of a planet yields an elliptic orbit, but we can't prove that the model pertains to actual planets (in fact, it does not).

By comparison, Watts and Strogatz's explanation of the small world phenomenon may seem less satisfying. First, the model is more abstract, which is to say less realistic. Second, the results are generated by simulation, not by mathematical analysis. Finally, the results seem less like a proof and more like an example.

Many of the models in this book are like the Watts and Strogatz model: abstract, simulation-based and (at least superficially) less formal than conventional mathematical models. One of the goals of this book is to consider the questions these models raise:

- What kind of work can these models do: are they predictive, or explanatory, or both?
- Are the explanations these models offer less satisfying than explanations based on more traditional models? Why?
- How should we characterize the differences between these and more conventional models? Are they different in kind or only in degree?

Over the course of the book I will offer my answers to these questions, but they are tentative and sometimes speculative. I encourage you to consider them skeptically and reach your own conclusions.

Chapter 4

Scale-free networks

4.1 Zipf's Law

Zipf's law describes a relationship between the frequencies and ranks of words in natural languages¹. The “frequency” of a word is the number of times it appears in a body of work. The “rank” of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.

Specifically, Zipf's Law predicts that the frequency, f , of the word with rank r is:

$$f = cr^{-s}$$

where s and c are parameters that depend on the language and the text.

If you take the logarithm of both sides of this equation, you get:

$$\log f = \log c - s \log r$$

So if you plot $\log f$ versus $\log r$, you should get a straight line with slope $-s$ and intercept $\log c$.

Exercise 4.1 Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency. You can test it by downloading an out-of-copyright book in plain text format from gutenberg.net. You might want to remove punctuation from the words.

If you need some help getting started, you can download greenteapress.com/compmo/Hist.py, which provides an object named `Hist` that you might find useful.

Plot the results and check whether they form a straight line. Can you estimate the value of s ?

You might want to use the `pylab` package, which provides easy-to-use functions to generate a variety of plots. In particular, `pylab.loglog` plots a series of points on a log-log scale. `pylab` is part of a

¹See wikipedia.org/wiki/Zipf's_law

larger package called `matplotlib`; it is not included in all Python distributions, so you might have to install it.

Here is an example that generates a simple graph:

```
import pylab

# compute lists of x and y values
xs = range(0, 10)
ys = [x**2 for x in xs]

# plot the data
pylab.plot(xs, ys, '-ro')

# set title and labels
pylab.title('Parabola')
pylab.xlabel('x')
pylab.ylabel('y = x**2')

# show the plot
pylab.show()
```

`pylab.plot` takes two lists as parameters. If you have a list of pairs instead, you can use `zip` and the scatter operator to “transpose” it. For example, if `t` is a list of pairs, `zip(*t)` returns a pair of lists.

The most common gotcha with `pylab` is that you have to call `pylab.show` to see the plot. You can read more about it at matplotlib.sourceforge.net.

You can download my solution from greenteapress.com/compmod/Zipf.py

4.2 Cumulative distributions

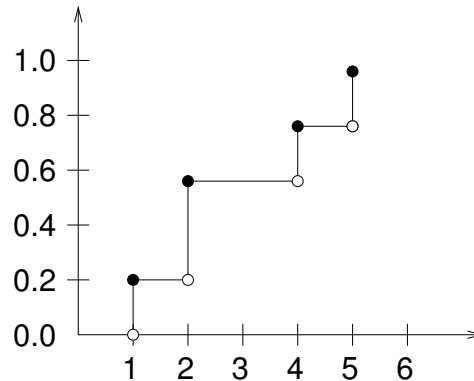
A distribution is a statistical description of a set of values. For example, if you collect the population of every city and town in the U.S., you would have a set of about 14,000 integers.

The simplest description of this set is a list of numbers, which would be complete but not very informative. A more concise description would be a statistical summary like the mean and variation, but that is not a complete description because there are many sets of values with the same summary statistics.

One alternative is a histogram, which divides the range of possible values into “bins” and counts the number of values that fall in each bin. Histograms are common, so they are easy to understand, but they have some drawbacks: the primary one is that it is tricky to get the bin size right. If the bins are too small, then the number of values in each bin is also small, so the histogram doesn’t give much insight. If the bins are too large, they lump together a wide range of values, which obscures details that might be important.

A better alternative is a **cumulative distribution**, which shows the percentage of values less than or equal to x for a range of x that sweeps from the smallest value in the set to the highest.

For example, the following figure shows the cumulative distribution function (CDF), for the values (1,2,2,4,5).



The range of the CDF is always from 0 to 1. The values on the y-axis are sometimes called **percentiles**, although strictly-speaking the range of a percentile is from 0 to 100. The values on the x-axis are called **quantities**, or sometimes “quantiles.”

The filled and empty circles indicate the value of the function at integral values: for example, $cdf(1) = 0.2$ because 20% of the values in the set are less than or equal to 1.

Exercise 4.2 Write a definition for a class named `Dist` that provides the following methods:

`__init__`: Takes a histogram object similar to the one defined in `greenteapress.com/compmo/Hist.py` and builds a representation of the distribution.

For example, to make a distribution of the values (1,2,2,4,5) you would run:

```
h = Hist([1,2,2,4,5])
d = Dist(h)
```

percentile: Takes a quantity and returns the corresponding percentile. For example, given the previous values, `d.percentile(2)` should return 0.6.

print_cdf: Print the quantities in the distribution and their corresponding percentiles, with two lines per quantity. For the previous distribution, the output should be:

```
1 0.0
1 0.2
2 0.2
2 0.6
4 0.6
4 0.8
5 0.8
5 1.0
```

plot_cdf: Use `pylab` to plot the distribution in the style of the figure above. You should plot two points per quantity to make a stair-step pattern. You don’t have to plot the filled and empty circles, but you can.

You should give some thought to choosing a data structure to represent a distribution. `percentile` should be $O(\log n)$, where n is the number of *unique* quantities in the distribution. The other methods should be linear in n .

You can download my solution from greenteapress.com/compmod/Dist.py

4.3 Closed-form distributions

The distributions we have seen so far are sometimes called **empirical distributions** because they are based on a dataset that comes from some kind of empirical observation.

An alternative is what I will call a **closed-form distribution**, which is characterized by a CDF that can be expressed as a closed-form function. Some of these distributions, like the Gaussian² or normal distribution, are well known, at least to people who have studied statistics. Many real world phenomena can be approximated by closed-form distributions, which is why they are useful.

For example, if you observe a mass of radioactive material with an instrument that can detect decay events, the distribution of times between events will most likely fit an exponential distribution. The same is true for any series of events where an event is equally likely at any time.

The CDF of the exponential distribution is:

$$cdf(x) = 1 - e^{-\lambda x}$$

The parameter, λ , determines the mean and variance of the distribution. This equation can be used to derive a simple visual test for whether a dataset can be well approximated by an exponential distribution. All you have to do is plot the **complementary distribution** on a log-y scale.

The complementary distribution (CCDF) is just $1 - cdf(x)$; if you plot the complementary distribution of a dataset that you think is exponential, you expect to see a function like:

$$y = 1 - cdf(x) \sim e^{-\lambda x}$$

If you take the log of both sides of this equation, you get:

$$\log y \sim -\lambda x$$

So on a log-y scale the CCDF should look like a straight line with slope $-\lambda$.

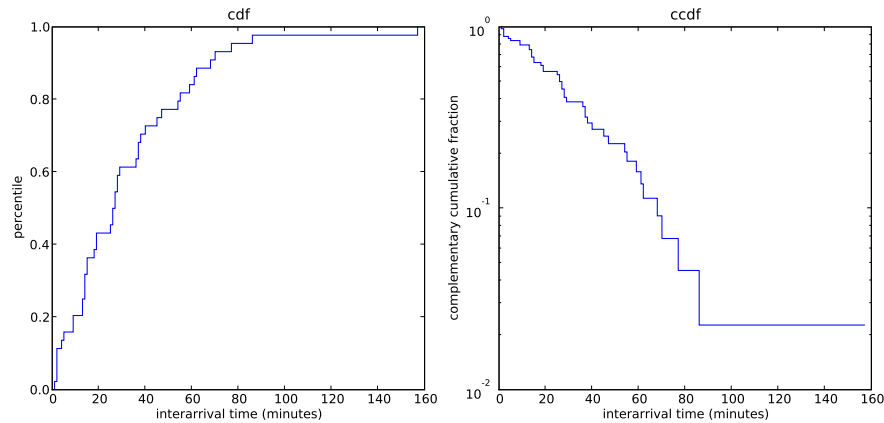
As an example, I will analyze the time of birth for 44 babies born in a 24-hour period in a hospital in Brisbane, Australia.³

Using the list of birth times, I computed the “interarrival times;” that is, the times between successive births. The following figure shows the CCDF of these values. The graph on the right is on a

²This is not a perfect example, since the CDF of a Gaussian is an integral with no closed-form solution.

³This example is based on information and data from Dunn, “A Simple Dataset for Demonstrating Common Distributions,” *Journal of Statistics Education* v.7, n.3 (1999). You can download a file with this data from greenteapress.com/compmod/babyboom.dat.

log-y scale. It is not particularly straight, which suggests that the exponential model is only an approximation of the observed process. Most likely the underlying assumption—that a birth is equally likely at any time of day—is not exactly true.



Exercise 4.3 In your `Dist` class, add a method called `plot_ccdf` that plots the complementary CDF of the distribution.

Exercise 4.4 Use the function `expovariate` in the `random` module to generate 44 values from an exponential distribution with mean 32.6. Plot the CCDF on linear and log-y scales and compare it to the figure above.

4.4 Pareto distributions

The Pareto distribution is named after the economist Vilfredo Pareto, who used it to describe the distribution of wealth (see wikipedia.org/wiki/Pareto_distribution). Since then, people have used it to describe a variety of phenomena in the natural and social sciences, including sizes of cities and towns, sand particles and meteorites, forest fires and earthquakes.

The Pareto distribution is characterized by a CDF with the following functional form:

$$1 - \left(\frac{x}{x_m} \right)^{-\alpha}$$

The parameters x_m and α determine the location and shape of the distribution. x_m is the minimum possible quantity.

Values from a Pareto distribution often have these properties:

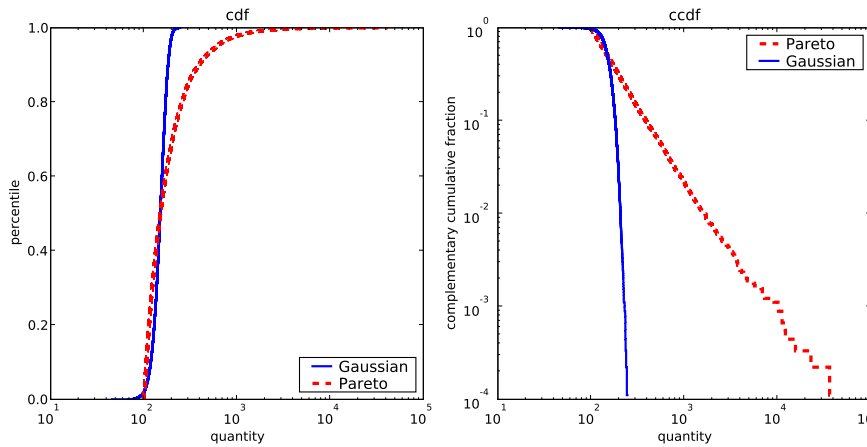
Long tail: Pareto distributions contain many small values and a few very large ones.

80/20 rule: The large values in a Pareto distribution are so large that they add up to a disproportionate share of the total. In the context of wealth, the 80/20 rule says that 20% of the people own 80% of the wealth.

Scale free: Short-tailed distributions are centered around a typical size, which is called a “scale.” For example, the great majority of adult humans are between 100 and 200 cm in height, so we could say that the scale of human height is a few hundred centimeters. But for long-tailed distributions, there is no similar range (bounded by a factor of two) that contains the bulk of the distribution. So we say that these distributions are “scale-free.”

To get a sense of the difference between the Pareto and Gaussian distributions, imagine what the world would be like if the distribution of human height were Pareto. Using the same minimum height, $x_m = 100$ cm, we could choose $\alpha = 1.7$, so that the median height is about 150 cm.

Here are empirical CDFs for samples with $n = 9000$ drawn from these two distributions:



The figure on the left is on a log- x scale. It shows that the median and lower bounds of the distributions are the same, but the Pareto distribution has a longer tail. The figure on the right is on a log-log scale. The Pareto distribution is a straight line except in the extreme tail, where it is noisy because of the limitations of a finite sample.

If you generate 6 billion random values from the Gaussian distribution shown in the figure, you would guess that the tallest person in the world is about 255 cm. In fact, the tallest currently living person, Leonid Stadnyk, is 259 cm.

But if you generate 6 billion random values from the Pareto distribution, you will see that the tallest person in the world could easily be taller than 80.5 km, which would technically make him or her an astronaut, at least in the United States (see wikipedia.org/wiki/Earth's_atmosphere). That's what it means to be scale-free!

There is a simple visual test that can indicate whether an empirical distribution is well-characterized by a Pareto distribution: on a log-log scale, the CCDF looks like a straight line. The derivation is similar to what we saw in the previous section.

The equation for the CCDF is:

$$y = 1 - cdf(x) \sim \left(\frac{x}{x_m} \right)^{-\alpha}$$

Taking the log of both sides yields:

$$\log y \sim -\alpha(\log x - \log x_m)$$

So if you plot $\log y$ versus $\log x$, it should look like a straight line with slope $-\alpha$ and intercept $-\alpha \log x_m$.

Exercise 4.5 The Weibull distribution is a generalization of the exponential distribution that comes up in failure analysis (see wikipedia.org/wiki/Weibull_distribution).

Can you find a transformation that makes a Weibull distribution look like a straight line? What do the slope and intercept of the line indicate?

Use `random.weibullvariate` to generate a sample from a Weibull distribution and use it to test your transformation.

Exercise 4.6 The distribution of populations for cities and towns has been proposed as an example of a real-world phenomenon that can be described with a Pareto distribution.

The U.S. Census Bureau publishes data on the population of every incorporated city and town in the United States. I have written a small program that downloads this data and converts it into a convenient form. You can download it from greenteapress.com/compmud/populations.py.

1. Read over the program to make sure you know what it does and then write a program that computes and plots the distribution of populations for the 14593 cities and towns in the dataset.
2. In your `Dist` class, write a function called `quantile` that takes a percentile as a parameter and returns the corresponding quantity. What is the median size in this distribution (the quantity that corresponds to the 50th percentile)? What are the 25th and 75th percentiles?
3. Plot the CDF on linear and log- x scales so you can get a sense of the shape of the distribution. Then plot the CCDF on a log-log scale to see if it has the characteristic shape of a Pareto distribution.
4. Use your plot to estimate values for the parameters x_m and α . Then use the function `paretovariate` in the `random` module to generate 14593 values from a Pareto distribution with the parameters you estimated in the previous problem. Plot the CCDF on a log-log scale and compare it the CCDF of the observed data.

What conclusion do you draw about the distribution of sizes for cities and towns?

4.5 Barabási and Albert

In 1999 Barabási and Albert published a paper in *Science*, “Emergence of Scaling in Random Networks,” that characterizes the structure (also called “topology”) of several real world networks, including graphs that represent the interconnectivity of movie actors, world-wide web (WWW) pages, and elements in the electrical power grid in the western United States.

They measure the degree (number of connections) of each node and compute $P(k)$, the probability that a vertex has degree k ; then they plot $P(k)$ versus k on a log-log scale. The tail of the plot fits a

straight line, so they conclude that it obeys a **power law**: as k gets large, $P(k)$ is asymptotic to $k^{-\gamma}$, where γ is a parameter that determines the rate of decay.

They also propose a model that generates random graphs with the same property. The essential features of the model, which distinguish it from the Erdős-Rényi model and the Watts-Strogatz model, are:

Growth: Instead of starting with a fixed number of vertices, Barabási and Albert start with a small graph, add vertices over time, and characterize the structure of the graph as the number of vertices grows.

Preferential attachment: When a new edge is created, it is more likely to connect to a vertex that already has a large number of edges. This “rich get richer” effect is characteristic of the growth patterns of some real-world networks.

Finally, they show that graphs generated by this model have a distribution of degrees that obeys a power law. Graphs that have this property are sometimes called **scale-free networks** (see wikipedia.org/wiki/Scale-free_network), but the name can be confusing because it is the distribution of degrees that is scale-free.

In order to maximize confusion, distributions that obey the power law are also sometimes called **scaling distributions** because they are invariant under a change of scale. That means that if you change the units the quantities are expressed in, the slope parameter, γ , doesn't change. You can read wikipedia.org/wiki/Power_law for the details, but it is not important for what we are doing here.

Exercise 4.7 This exercise asks you to make connections between the Watts-Strogatz (WS) and Barabási-Albert (BA) models:

1. Read Barabási and Albert's paper and implement their algorithm for generating graphs. See if you can replicate their Figure 2(A), which shows $P(k)$ versus k for a graph with 150 000 vertices.
2. Use the WS model to generate the largest graph you can in a reasonable amount of time. Plot $P(k)$ versus k and see if you can characterize the rate of decay.
3. Use the BA model to generate a graph with about 1000 vertices and compute the characteristic length and clustering coefficient as defines in the Watts and Strogatz paper. Do scale-free networks have the characteristics of a small-world graph?

4.6 Zipf, Pareto and power laws

At this point we have seen three phenomena that yield a straight line on a log-log plot:

Zipf law: A Zipf plot shows frequency as a function of rank.

Pareto CCDF: A CCDF shows percentile as a function of quantile.

Power law distribution: A power law plot shows frequency (or probability) as a function of quantile.

The similarity in these plots is not a coincidence; these visual tests are closely related.

Starting with a power-law distribution, we have:

$$P(k) \sim k^{-\gamma}$$

$P(k)$ is the probability that a random variable X equals k , so the cumulative distribution of X is:

$$cdf(x) = Pr\{X \leq x\} = \sum_{k=0}^x P(k)$$

For large values of k we can approximate the summation with an integral:

$$\sum_{k=0}^x k^{-\gamma} \sim \int_{k=0}^x k^{-\gamma} = \frac{1}{\gamma-1} (1 - x^{-\gamma+1})$$

To make this a proper CDF we could normalize it so that it goes to 1 as x goes to infinity, but that's not necessary, because all we need to know is:

$$cdf(x) \sim 1 - x^{-\gamma+1}$$

Which shows that the distribution of x is asymptotic to a Pareto distribution with $x_m = 1$ and $\alpha = \gamma - 1$.

Similarly, if we start with a straight line on a Zipf plot, we have⁴

$$f = cr^{-s}$$

Where f is the frequency of the word with rank r . Inverting this relationship yields:

$$r = (f/c)^{-1/s}$$

Now subtracting 1 and dividing through by the number of different words, n , we get

$$\frac{r-1}{n} = \frac{(f/c)^{-1/s}}{n} - \frac{1}{n}$$

Which is only interesting because if r is the rank of a word, then $(r-1)/n$ is the fraction of words with lower ranks, which is the fraction of words with higher frequency, which is the CCDF of the distribution of frequencies:

$$ccdf(x) = Pr\{X > x\} = \frac{(f/c)^{-1/s}}{n} - \frac{1}{n}$$

⁴This derivation follows Adamic, "Zipf, power law and Pareto—a ranking tutorial," available at www.hpl.hp.com/research/idl/papers/ranking/ranking.html

Where X is a random variable drawn from the distribution of frequencies. To characterize the asymptotic behavior for large n we can ignore c and $1/n$, which yields:

$$ccdf(x) \sim f^{-1/s}$$

Which shows that if a set of words obeys Zipf's law then the distribution of their frequencies is asymptotic to a Pareto distribution with $x_m = 1$ and $\alpha = 1/s$.

So the three visual tests are mathematically equivalent; a dataset that passes one test will pass all three. But as a practical matter, the power law plot is noisier than the other two, because it is the (discrete) derivative of the CCDF.

The Zipf and CCDF plots are more robust, but Zipf's law is only applicable to discrete data (like words), not continuous quantities. CCDF plots work with both.

For these reasons—robustness and generality—I recommend using CCDFs exclusively.

Exercise 4.8 The Internet Movie Database (IMDb) is an online collection of information about movies. Their database is available in plain text format, so it is reasonably easy to read from Python. For this exercise, the files you need are `actors.list.gz` and `actresses.list.gz`; you can download them from www.imdb.com/interfaces#plain.

I have written a program that parses these files and splits them into actor names, movie titles, etc. You can download it from greenteapress.com/compmdb/imdb.py.

If you run `imdb.py` as a script, it reads `actors.list.gz` and prints one actor-movie pair per line. Or, if you import `imdb` you can use the function `process_file` to, well, process the file. The arguments are a filename, a function object and an optional number of lines to process. Here is an example:

```
import imdb

def print_info(actor, date, title, role):
    print actor, date, title, role

imdb.process_file('actors.list.gz', print_info)
```

When you call `process_file`, it opens `filename`, reads the contents, and calls `print_info` once for each line in the file. `print_info` takes an actor, date, movie title and role as arguments and prints them.

1. Write a program that reads `actors.list.gz` and `actresses.list.gz` and builds a database that maps from each actor to a list of his or her films.
2. Two actors are “costars” if they have been in at least one movie together. Process the database you built in the previous step and build a second database that maps from each actor to a list of his or her costars.
3. Write a program that can play the “Six Degrees of Kevin Bacon” game, which you can read about at wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon.

4. Estimate the characteristic length and clustering coefficient of the costar graph. Is it a small world after all?
5. Plot the CCDF of the distribution of degrees in the graph on a log-log scale. Is it scale-free?

