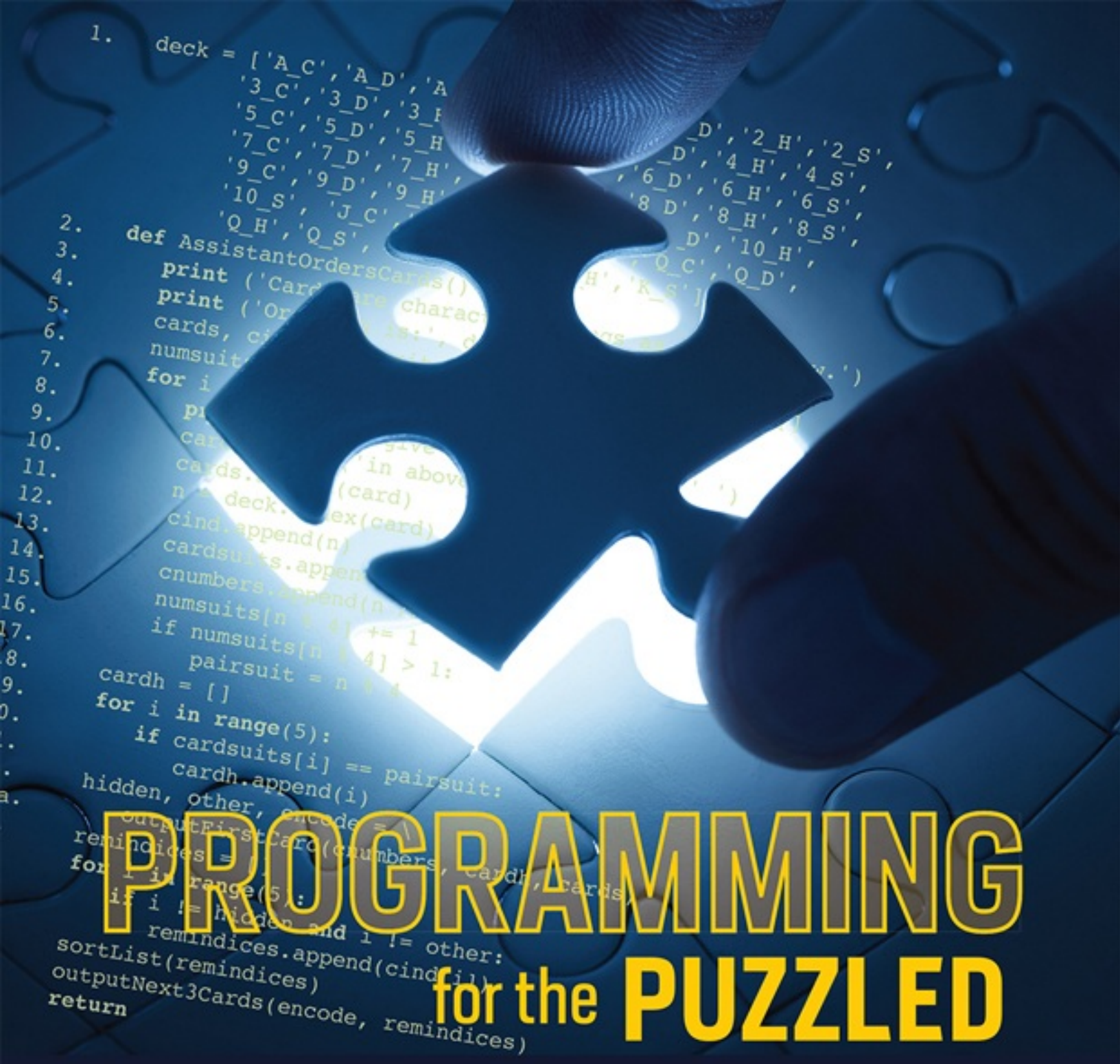


PROGRAMMING for the PUZZLED

LEARN TO PROGRAM WHILE SOLVING PUZZLES

SRINI DEVADAS

Programming for the Puzzled

Learn to Program While Solving Puzzles

Srini Devadas

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Times Roman by Westchester Publishing Services. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Devadas, Srinivas, author.
Title: Programming for the puzzled : learn to program while solving puzzles / Srini Devadas.
Description: Cambridge, MA : The MIT Press, [2017] | Includes bibliographical references and index.
Identifiers: LCCN 2017010148 | ISBN 9780262534307 (pbk. : alk. paper)
Subjects: LCSH: Computer programming--Computer-assisted instruction. | Mathematical recreations--Data processing. | Puzzles--Data processing. | Computer games.
Classification: LCC QA76.6 .D485 2017 | DDC 005.10285--dc23 LC record available at https://lccn.loc.gov/2017010148

Contents

Preface

Puzzles are wonderfully recreational. The best puzzles have nonobvious solutions that need an “Aha moment” to be discovered. An algorithmic puzzle is a puzzle whose solution is an algorithm—a set of steps that can be mechanized. Algorithms can be described in English or any other natural language, though for greater precision they are often described in “pseudocode.” Pseudocode is called that for a reason—it is not detailed enough that it can be run on a computer, unlike code written in a programming language.

Computer programming is a livelihood for a growing number of people in the world. To learn programming, one first learns basic programming constructs like assignment statements and control loops through simple examples, and programming exercises often involve translating the pseudocode of an algorithm into code in the programming language being learned. Programmers benefit from the same analytical skills that puzzle-solving requires. These skills are required when translating specifications into programming constructs, as well as discovering errors in early versions of code, called the debugging process.

While teaching programming at MIT to freshmen and sophomores over decades, it became clear to me that students are strongly motivated by applications. Few want to program for programming’s sake. Puzzles are some of the coolest applications around—they have the advantage of being easy to describe and attention-grabbing. The latter is particularly important today, when lecturers have to compete for attention with Snapchat, Facebook, and Instagram. And I discovered, as others have before me, that the best way of putting students to sleep is to describe programming syntax or semantics ad nauseum, meaning for more than a couple of minutes!

This book is my attempt at teaching programming by building a bridge

between the recreational world of algorithmic puzzles and the pragmatic world of computer programming. I assume a rudimentary grasp of programming concepts that can be obtained from introductory or AP computer science classes in high school, or classes like MITx/edX 6.0001x.

Each lesson in the book starts with the description of a puzzle. Many of these puzzles (chapters) are popular ones that have been described in several publications and websites, with many variations. After a failed attempt or two at solving the puzzle, we arrive at an Aha moment—a search strategy, data structure, or mathematical fact—and voilà, the solution presents itself. Sometimes there is an obvious "brutish" way of solving the puzzle, and I explain the associated algorithm and code before deeming it a "failure." And then there is an insight that leads to a more elegant and efficient solution.

The solution to the puzzle is the *specification* of the code we need to write. As you read, you will learn what the code is supposed to do before you see the code. This is a powerful pedagogical philosophy that I believe in, since it decouples understanding the code's functionality from understanding programming language syntax and semantics. Syntax and semantics required to understand the code are explained on a "pay as you go" basis.

Going from the physical world of the puzzle to the computer world of the program is fun but not always smooth. In some cases you will have to pretend in the computer world that some operation is not efficient because it is not efficient in the physical world. I've tried to minimize that in the book, but have not eliminated it entirely. I trust this will not cause confusion, and I point it out in the very few instances it occurs.

You could read and use this book in many ways. If you are only interested in puzzles and their solutions, you can stop after you come up with the solution or after you have read the provided solution. I hope you don't stop there, because showing how to take the described solution and turn it into executable code was a primary purpose in my writing the book. Reading an entire puzzle will give you a good sense of what it takes to produce a useful program that anyone can run and use for themselves. I tried hard to ensure that my descriptions of Python syntax and semantics were self-contained, but if you have questions about Python syntax, semantics, and libraries, python.org is an excellent resource, and the edX/MITx course 6.0001x is a great introduction to programming in Python.

If you can install and run Python on your machine, you will get a lot more from this book. You can do this by visiting https://www.python.org/downloads/. The code for all the puzzle solutions in the book is conveniently downloadable

from the book's MIT Press website or from https://mitpress.mit.edu/books/programming-puzzled. The code has been tested for Python version 2.7 and above, including Python 3.x releases.[1] Of course, you are welcome to ignore the website and write your own code that solves the puzzle. You can run the programs downloaded from the book's website, or the programs that you have written for different inputs from the examples in the book, and I strongly encourage you to do so. I don't make any guarantees about there being no bugs in the code, though I did try to get rid of all of them. Be warned that the provided code makes assumptions about its inputs corresponding to the puzzle statement and doesn't behave gracefully when it receives inputs it does not expect. Adding these checks would have cluttered the code. A good way of deepening your understanding of programming is to augment each puzzle's code to explicitly check for malformed input.

At the end of each puzzle I have a few programming exercises. These exercises vary in challenge level and the amount of code that needs to be written. Doing the exercises for each puzzle will help you get the most of out of this book. You will have to understand the code associated with the puzzle well enough to be able to modify it or augment the code's functionality. A few of the exercises in the book relate to including checks for malformed input. The exercises marked **Puzzle Exercise** require significant code-writing or restructuring of the provided puzzle code. Some of these could be viewed as advanced puzzles related to the puzzle described. Solutions to the exercises and puzzles are not provided in the book, but are available to instructors from the MIT Press book website.

I am a firm believer in learning by doing—if you successfully do all the exercises by yourself, you will be well on your way to becoming a computer scientist! I wish you the best of luck on your journey.

Note

1. In the case of one puzzle, "You Can Read Minds (With a Little Calibration)," the code is different for Python 2.x and Python 3.x, but for all other puzzles, the same code should work.

Acknowledgments

I wrote this book while on sabbatical at my alma mater, the University of California, Berkeley. I thank Professor Robert Brayton for kindly letting me use his office in Cory Hall, where I wrote a large part of the book. I am grateful to my Berkeley hosts, Sanjit Seshia, Kurt Keutzer, and Dawn Song for a most enjoyable and productive sabbatical.

I taught software engineering for the first time with Daniel Jackson—a course that used Java as the programming language. I've been and continue to be greatly influenced by Daniel's views on programming languages and software engineering. Our collaboration on JavaScript accelerator workshops included puzzles like how to compute postage in different ways to teach basic programming concepts.

I first taught Introduction to Computer Science and Programming with John Guttag—a course taken largely by students who are not computer science majors. John's enthusiasm is infectious, and it got me hooked into teaching introductory programming in Python. At least one of his examples—bisection search to find square roots—found its way into this book.

I used some of the puzzles in this book in lectures in Fundamentals of Programming, a course that I codeveloped with Adam Chlipala and Ilia Lebedev in 2015–2016. With code written for class lectures and assignments, many people contributed from draft to the "production" version. I especially note the contributions of Yuqing Zhang, who wrote code for "Tile That Courtyard, Please," Rotem Hemo, who wrote code for "Counting the Ways You Can Count Change," and Jenny Ramseyer, who wrote code for "Questions Have a Price." About four hundred students took the course in Spring 2017, and teaching material from this book with Duane Boning, Adam Hartz, and Chris Terman was

a most enjoyable experience.

The puzzle “You Can Read Minds (with a Little Calibration)” has been a staple in the course Mathematics for Computer Science. I am not sure who invented it, but I thank Nancy Lynch for introducing it to me and being “assistant” to my “magician” during my first nerve-racking lecture demonstration over fifteen years ago. The description of the puzzle comes from notes for the class written by instructors Eric Lehman, Tom Leighton, and Albert Meyer.

Kaveri Nadhamuni helped with the code in several puzzles, including “Find That Fake,” “Guess Who Isn’t Coming to Dinner,” “America’s Got Talent,” and “Greed Is Good.” Eleanor Boyd tested the puzzle “Find That Fake” in the Girls Who Code program and provided valuable feedback.

Ron Rivest suggested optimizations and generalizations for numerous puzzles, including “You Will All Conform,” “The Best Time to Party,” “Please Do Break the Crystal,” “Guess Who Isn’t Coming to Dinner,” “Anagramania,” and “Memory Serves You Well.”

Billy Moses carefully read through the book and suggested numerous improvements.

My facility with algorithms has improved thanks to teaching the classes Introduction to Algorithms, as well as Design and Analysis of Algorithms, with my colleagues Costis Daskalakis, Erik Demaine, Manolis Kellis, Charles Leiserson, Nancy Lynch, Vinod Vaikuntanathan, Piotr Indyk, Ron Rivest, and Ronitt Rubinfeld. My knowledge of software engineering and programming languages has grown through teaching software classes with Saman Amarasinghe, Adam Chlipala, Daniel Jackson, John Guttag, and Martin Rinard. I am grateful to these talented colleagues.

Victor Costan and Staphany Park made it easy for me to focus on course content by creating wonderful automated grading infrastructure for Fundamentals of Programming and Introduction to Algorithms.

MIT Press sent a version of this book to three anonymous reviewers. I thank them for their careful reading of the manuscript, for numerous excellent suggestions to improve the book, and, of course, for recommending publication. I hope that if and when they read the published book, they will be happy with my efforts in incorporating their valuable feedback.

At MIT, I have been constantly encouraged in my educational endeavors by departmental as well as laboratory administration and leadership. I thank Duane Boning, Anantha Chandrakasan, Agnes Chow, Bill Freeman, Denny Freeman,

Eric Grimson, John Guttag, Harry Lee, Barbara Liskov, Silvio Micali, Rob Miller, Daniela Rus, Chris Terman, George Verghese, and Victor Zue for their support over many years.

Marie Lufkin Lee, Christine Savage, Brian Buckley, and Mikala Guyton capably handled the reviewing, editing, and production of the book. Robie Grant created the index. I thank them for their efforts.

My wife Lochan suggested the title for the book. My daughters Sheela and Lalita were my first “customers” and helped shape the book. This book is dedicated to all three of them.

1

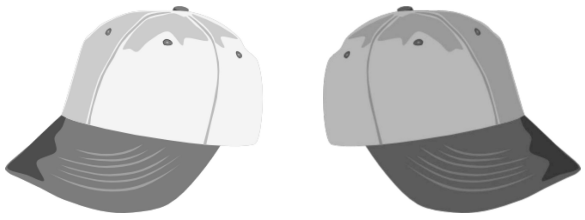
You Will All Conform

When people are free to do as they please, they usually imitate each other.
—Eric Hoffer

> Programming constructs and algorithmic paradigms covered in this puzzle: Lists, tuples, functions, control flow (including **if** statements and **for** loops), and **print** statements.

Let’s say we have a whole bunch of people in line waiting to see a baseball game. They are all hometown fans and each person is wearing a team cap. However, not all of them are wearing the caps in the same way—some are wearing their caps in the normal way, and some are wearing them backward.

People have different definitions of normal and backward, but you think the cap on the left (see below) is being worn normally, and the one on the right is being worn backward.



You are the gatekeeper and can only let the fans into the stadium if the entire group has their caps on the same way—either all forward or all backward. Because everyone has different definitions of forward and backward, you can’t

tell them to wear their caps forward or backward. You can only tell them to flip their caps. The good news is that each person knows what position they are in the line, with the first person having position 0 and the last one position $n - 1$. You can say things like:

Person in position i*, please flip your cap.*
People in positions i *through* j *(inclusive), please flip your caps.*

But what you would like to do is minimize the number of requests you have to shout, to save your voice. Here is an example:

We have thirteen people standing in line in positions 0 through 12, inclusive. Since there are six people with caps on forward, we could shriek out six commands, for example:

Person in position 0, please flip your cap.

and repeat for the people in positions 1, 5, 9, 10, and 12. A voice-saving measure would exploit the second type of command and only yell out four commands:

People in positions 0 through 1, please flip your caps.
Person in position 5, please flip your cap.
People in positions 9 through 10, please flip your caps.
Person in position 12, please flip your cap.

and this will get everyone to have caps on backward.

However, in this example, we can do one better. If we scream out:

People in positions 2 through 4, please flip your caps.
People in positions 6 through 8, please flip your caps.
Person in position 11, please flip your cap.

this will get everyone with caps on forward.

How can we generate the minimum number of commands? A harder question: Can you think of a way of generating the commands as you walk down the line for the *first* time?

Think about it before moving on.

Finding Sequences of Like-Minded People

Assume we are given a list of cap orientations corresponding to the set of people waiting in line. We can compute a list of "intervals" corresponding to groups of contiguous people wearing caps the same way. An interval can be represented using its start and end points, say, [*a*, *b*], with $a < b$. This means that all positions between *a* and *b*, including *a* and *b*, are in the interval.

Each interval is labeled a forward or backward interval. So an interval has three properties: a start number or position, an end position, and a label signifying whether it is forward or backward.

How to compute the list of intervals? The key observation is that an interval ends and another begins when we see a change in orientation. And of course the first interval has start position 0. In our example, repeated below,

we see that position 0 has a forward orientation. Walking down the list, we see that position 1 is forward as well. But position 2 is backward; the orientation has flipped. This means the first interval has ended on the *previous* cap. The first interval is [0, 1] with an orientation of forward. Not only that, we have also determined the start position of the second interval, which is 2. Now we are in exactly the same situation as we were at the beginning: We know the start position of the current interval, and we need to figure out where it ends.

Going on in this way, we will generate [0, 1] forward, [2, 4] backward, [5, 5] forward, [6, 8] backward, [9, 10] forward, [11, 11] backward, and [12, 12] forward. The last interval didn't end because the orientation flipped, it ended because we ran out of people. (It will be important to handle this case properly in the code—watch out for it!)

Our first algorithm counts the number of forward intervals and the number of backward intervals, and it picks a particular orientation to flip depending on which set of intervals is smaller. In our example, we have four forward intervals and three backward intervals, so we want to ask the backward-cap people in the three backward intervals to flip their caps.

In the worst case, people have alternating forward and backward cap orientations, and given *n* people, assuming *n* is even, we will have $n/2$ forward intervals and $n/2$ backward intervals. In the worst case you have to yell out $n/2$ commands. If *n* is odd, we will have one more forward interval than backward, or vice versa.

What our algorithm does is conflate contiguous people with the same cap

orientations by grouping them into an interval. After the intervals have been determined, we have exactly the same situation as the alternating people above —there will be *m* forward intervals interspersed between *m*, *m* − 1, or *m* + 1 backward intervals. The best we can do is to choose the orientation that results in fewer commands. We can't do better than this algorithm.

Strings, Lists, and Tuples

In this book you will see strings, lists, tuples, sets, and dictionaries, which are all data structures provided by Python. Here's a brief overview of what you can do with these data structures.

Characters are single symbols, for example, `'a'`, `'A'`. Strings are sequences of characters, for example, `'Alice'`, or single characters, for example, `'B'`. A string can be represented with single quotes or with double quotes, for example, `"A"`, `"Alice"`. You can access individual characters in strings. For example, if `s = 'Alice'`, `s[0]` evaluates to `'A'`, and `s[len(s)-1] = 'e'`. The built-in function **len** returns the length of the argument string (or list). `len(s)` returns `5`. Strings cannot be modified. The assignment `s[0] = 'B'` will result in an error. However, you can create new strings by manipulating old ones. For example, given `s = 'Alice'`, you can write `s = s + 'A'` to get a new string `'AliceA'` referred to by `s`.

Lists in Python can be thought of as arrays or sequences of elements. They can contain numbers or strings or other lists. For example, let `L = [1, 2, 'A', [3, 4]]`. Then `L[0]` evaluates to `1`, `L[len(L)-1]` evaluates to `[3, 4]`, and `L[3][0]` will return `3`. Lists can be modified. `L[3][1] = 5` will mutate `L` to `[1, 2, 'A', [3, 5]]`.

Tuples are similar to lists but cannot be modified. Given `T = (1, 2, 'A', [3, 4])`, `T[3] = [3, 5]` will crash but `T[3][0] = 4` will mutate the list element in `T` to produce `T = (1, 2, 'A', [4, 4])`. If we write `T = (1, 2, 'A', (3, 4))`, `T[3][0] = 4` will crash.

You will see more operations on strings, lists, and tuples when we present them in code examples. We will describe sets and dictionaries later in the book, when we need to use these more advanced data structures.

From Algorithm to Code

Now we are ready to present the code for our first strategy. We present the code in two parts, explaining each part immediately after the code fragment. In all the code shown in the book, Python keywords or reserved words are shown in bold. Do not use these words as variable or function names in code that you write.

```
1.  cap1 = ['F','F','B','B','B','F','B',
            'B','B','F','F','B','F']
2.  cap2 = ['F','F','B','B','B','F','B',
            'B','B','F','F','F','F']
```

Lines 1–2 simply correspond to input lists. The list `cap1` is the example described earlier with the pretty cap pictures. The list is a list of strings where the strings represent the cap orientation: `'F'` for forward and `'B'` for backward. Python allows us to declare a list across multiple lines—the enclosing `[]` tells it where the list begins and ends.

```
3.  def pleaseConform(caps):
4.      start = forward = backward = 0
5.      intervals = []
6.      for i in range(1, len(caps)):
7.          if caps[start] != caps[i]:
8.              intervals.append((start, i-1, caps[start]))
9.              if caps[start] == 'F':
10.                 forward += 1
11.             else:
12.                 backward += 1
13.             start = i
14.     intervals.append((start, len(caps)-1, caps[start]))
15.     if caps[start] == 'F':
16.         forward += 1
17.     else:
18.         backward += 1
19.     if forward < backward:
20.         flip = 'F'
21.     else:
22.         flip = 'B'
23.     for t in intervals:
24.         if t[2] == flip:
25.             print ('People in positions', t[0],
                       'through', t[1], 'flip your caps!')
```

Line 3 names the function and lists its arguments. The **def** keyword is used to define the function, and the strings enclosed within the parentheses are the arguments. Our function only takes the input list as an argument. We will be able to invoke it with any list, including `cap1` and `cap2`. As an example, we will show the result of executing `pleaseConform(cap1)` later. The function assumes that the input argument is a list of `'F'` and `'B'` strings. The list can be of any length but

cannot be empty.

Lines 4–5 initialize the variables used in the algorithm. Each interval is represented as a 3-tuple where the first two elements are numbers and the third is an 'F' or a 'B' string label. The first two numbers give the endpoints of the interval. The interval is closed on both sides, that is, it includes both endpoints. As mentioned earlier, tuples are similar to lists, but unlike lists you can't modify a tuple once you have created it. We could just as easily have used on line 7:

```
intervals.append([start, i-1, caps[start]])
```

that is, [] around the three variables rather than (), and the program would work just the same. The variable intervals is a list of interval tuples and is initialized to the empty list []. The variables forward and backward count the number of forward and backward intervals, respectively. These are initialized to 0.

Lines 6–13, beginning with the **for** loop, compute intervals. **len**(cap1) will return 13. Note that the list elements in caps are numbered 0 through 12, for example, caps[0] = 'F', and caps[12] = 'F'. (Accessing caps[13] will give an error.) The **range** keyword takes one, two, or three arguments. If we had **range**(**len**(caps)), the variable i would start from 0 and increment all the way to **len**(caps) - 1. On line 6, we see **range**(1, **len**(caps)), which means that variable i iterates from 1 to **len**(caps) - 1, incrementing by 1 each time—the same as if we wrote **range**(1, **len**(caps), 1). If we wrote **range**(1, **len**(caps), 2), we would increment i by 2 each iteration.

The variable start is crucial to determining intervals. Initially start is 0, and we iterate through the various caps[i] till we find a caps[i] that is different from caps[start]. This check is done by the **if** statement on line 7. If the predicate of the **if** statement caps[start] != caps[i] is **True**, then we have ended one interval and begun another at this i. The interval that has just been found starts with start and ends with i - 1. Once the interval is determined, it is appended as a 3-tuple to the list intervals, where the first item is the start position, the second is the end position, and the third item is the type of interval, 'F' or 'B'. Lines 9–12 increment the appropriate count of forward or backward intervals. Line 13 sets start to i since we are beginning a new interval starting with i.

Note that line 14 is outside the **for** loop—the indentation tells you that. Once the **for** loop completes execution, you may think we have computed all the intervals. Not true! The last interval has not been added to the list intervals. This is because we only add an interval when we realize that we are past it. This happens when we see a caps list entry that is different from caps[start]. But this doesn't happen for the last interval! For example, with cap1 as input, when i = 12,

we will see 'F' with start = 11, and then exit the loop. Similarly, for cap2, when i = 12, we will see 'F' with start = 9 and then exit the loop. So we have to add the final interval outside the loop. We do this in lines 14–18 the same way we did before.

Lines 19–22 determine whether the forward or backward intervals should be flipped. The smaller set is chosen. Then, in lines 23–25 we loop over the intervals. This **for** loop iterates over each interval t in intervals and prints the command for the chosen type of interval, either forward or backward. Each t is a 3-tuple with the information associated with each interval, and this is used to selectively print and to generate the start and end positions for each command. For a tuple t, t[0] is the start of the interval, t[1] is the end, and t[2] is the type. If we tried to set t[0] or t[1] or t[2] to anything, the program would crash, since tuples cannot be written. It is a good idea to use tuples rather than lists when you do not want to inadvertently change the value of elements in the list.

The **print** statement on line 25 prints out the commands. The **print** statement prints out strings interspersed with variable values. The strings and variables to be printed have to be enclosed in ().[1] Notice that we have split the **print** statement across two lines for readability, but Python will be able to parse the **print** statement properly since its arguments are enclosed in ().

Code Optimization

Inside every large program, there is a small program trying to get out.
Tony Hoare

Twenty-six lines of code isn't a large program, but the beauty of programming algorithmic puzzles is that optimizations can usually be made to shrink the code. Smaller programs are usually more efficient and have fewer bugs, though that is certainly not a hard-and-fast rule.

We had a special case associated with the end of the list, where in lines 14–18 we add the last interval. This is because we only add an interval when we see an entry different from caps[start]. To avoid the special case, all we have to do is introduce a statement between lines 5 and 6, namely:

```
5a.     caps = caps + ['END']
```

and simply eliminate lines 14–18. The statement above adds one more element to the list caps that is different from every other entry. The + operator on two lists concatenates them and produces a new list corresponding to the concatenation. This is why we need an encircling [] around 'END': because the + operator only

operates on two lists or two strings or two numbers, not a list and a string, or a string and a number. Adding an element means we will have one more iteration of the loop, and in this last iteration caps[start] != caps[i] will be **True** regardless of caps[start] being 'F' or 'B', leading to the addition of the last interval.

Finally, one of the nice side effects of this optimization is that the optimized program no longer crashes on the empty list as the original program did.

List Creation and Modification

We could have used caps.append('END') in line 5a to append a new string element to the list caps. However, this modifies the argument list caps, which is something we should avoid doing. Suppose we run the two different programs that follow:

```
1.    def listConcatenate(caps):
2.        caps = caps + ['END']
3.        print(caps)

4.    capA = ['F','F','B']
5.    listConcatenate(capA)
6.    print(capA)
```

and

```
1.    def listAppend(caps):
2.        caps.append('END')
3.        print(caps)

4.    capA = ['F','F','B']
5.    listAppend(capA)
6.    print(capA)
```

The first will print ['F','F','B','END'] and then ['F','F','B'] and the second will print ['F','F','B','END'] twice. The list concatenation operator + creates a new list, but append modifies the existing one. So in the append case, capA outside the procedure call has been modified.

Scoping

We would have exactly the same behavior for both programs if capA were replaced everywhere by caps. Let's look at the first program with capA replaced by caps.

```
1.    def listConcatenate(caps):
2.        caps = caps + ['END']
```

```
3.          print(caps)

4.  caps = ['F','F','B']
5.  listConcatenate(caps)
6.  print(caps)
```

This will print ['F','F','B','END'] and then ['F','F','B']. The variable caps outside the procedure listConcatenate has a different scope from the argument caps inside the function. The argument caps will point to a new list, that is, different memory ['F','F','B','END'] once the concatenation is performed because concatenation *creates* this new memory and copies the list elements into it. After the procedure completes execution, the argument caps and the new list disappear and cannot be accessed. The variable caps outside the procedure still points to list ['F','F','B'] in its original memory location, which was not modified.

Now, let's look at the append case.

```
1.  def listAppend(caps):
2.          caps.append('END')
3.          print(caps)

4.  caps = ['F','F','B']
5.  listAppend(caps)
6.  print(caps)
```

The scoping rules apply here as well. The argument caps is initially pointing to the single list ['F','F','B']. The append *modifies* this list to ['F','F','B','END']. So even after the procedure finishes execution and the argument variable caps disappears, the memory location has been modified with the additional element 'END'. We see the effect of this even outside the procedure since the variable caps outside the procedure is pointing to the same memory location.

If this description made your head hurt, please use different names for argument variables and variables outside the procedure that are passed to procedure calls or invocations.

An Algorithmic Optimization

Now, let's look at the harder question of how we can determine the minimum set of commands when we walk down the line the very first time.

Here's a hint. The numbers of forward and backward intervals differ by at most 1. Observe that if the first person has a particular orientation, say, forward, the number of forward intervals can never be smaller than the number of backward intervals. Similarly, if the first person has a backward orientation, the

number of backward intervals can never be smaller than the number of forward intervals.

Our first algorithm is two-pass in the sense that it first determines forward and backward intervals by iterating through the list (the first pass), then goes through the intervals to print the appropriate commands (the second pass).

Can you think of an algorithm that makes a single pass over the list to generate the minimum set of commands? Your algorithm should be implemented using a single loop.

A One-Pass Algorithm

We exploit the observation that the orientation of the first cap in the list tells us whether the set of forward or backward intervals corresponds to the minimum set of commands. This leads us to a one-pass algorithm, described in the following code.

```
1.    def pleaseConformOnepass(caps):
2.        caps = caps + [caps[0]]
3.        for i in range(1, len(caps)):
4.            if caps[i] != caps[i-1]:
5.                if caps[i] != caps[0]:
6.                    print('People in positions', i, end='')
7.                else:
8.                    print(' through', i-1, 'flip your caps!')
```

Line 2 adds an element to the list, but rather than adding an end element, it adds an element that is the same as the first element. During the loop iterations, the first time we encounter an element that is different from the first element (line 4), we *start* an interval. This means we are actually *skipping* the first interval. This is okay because of our observation that the first interval corresponds to an orientation that is not the sole optimum to generate commands for—at best it ties the opposite orientation in terms of number of commands required. When we encounter an element that is the same as the first element (line 7), we end the interval. The code prints out the commands as the intervals are constructed. Adding the extra element to the end of the list, equaling the first element on line 2, ensures that the last interval is printed in the case where the original last element was different from `caps[0]`.

The downside is that this code is not as easy to modify as the original code to do the exercises at the end of the chapter. It also needs to be augmented to not

crash on an empty list.

Applications

The motivation behind this particular puzzle is compression. The same amount of information associated with commands to single persons can be compressed into a smaller set of commands, each directed toward a contiguous group of people.

Data compression is an important application and is becoming even more important as we generate huge amounts of data on the Internet. Lossless data compression can be performed in many ways—one algorithm close to our puzzle in spirit is called run-length encoding. It is best described using a simple example. Suppose we have an alphabet string that has 32 characters:

```
WWWWWWWWWWWWWBBWWWWWWWWWWWWBBBBB
```

Using a simple algorithm, we can compress it to an alphanumeric string with 10 characters:

```
13W2B12W5B
```

In the above string, prefix numbers before each alphabet character tell us how many of these characters appear in sequence in the original string. The original string has 13 `W`'s, then 2 `B`'s, then 12 `W`'s, and finally 5 `B`'s, and we represent this information directly in the compressed string.

Run-length decoding is the process of decompressing `13W2B12W5B` into the original string.

While our first example resulted in significant compression, suppose we have a string like this one:

```
WBWBWBWBWB
```

Our run-length encoding scheme will naively produce a longer string, shown below.

```
1W1B1W1B1W1B1W1B1W1B
```

However, a smarter algorithm might produce something like this:

```
5(WB)
```

`()`'s can be understood to enclose the sequence that repeats. Compression utilities available in modern computers use algorithms that are related to these ideas.

Exercises

Exercise 1: One slightly annoying thing is that executing pleaseConform(cap1) prints:

```
People in positions 2 through 4 flip your caps!
People in positions 6 through 8 flip your caps!
People in positions 11 through 11 flip your caps!
```

It should print for the last command:

```
Person at position 11 flip your cap!
```

Modify the code so the commands sound more natural.

Exercise 2: Modify pleaseConformOnepass to print more natural commands as in exercise 1, and ensure that it does not crash on an empty list.

Hint: You will need to remember the beginning of the interval (and not print on line 6).

Puzzle Exercise 3: Suppose there are bareheaded people in the line. We'll represent them with the 'H' character. So for example we might have:

```
cap3 = ['F','F','B','H','B','F','B',
        'B','B','F','H','F','F']
```

We don't want to confuse the bareheaded people by telling them to flip their nonexistent caps and perhaps cause one of them to try to steal a cap from the person ahead in line. Therefore, we want to skip over all the 'B' positions. Modify pleaseConform so it generates a correct and minimal set of commands. For the above example it should generate:

```
Person in position 2 flip your cap!
Person in position 4 flip your cap!
People in positions 6 through 8 flip your caps!
```

Exercise 4: Write a program that performs simple run-length encoding, which converts a given string, for example, BWWWWWBWWWW, to the shorter 1B5W1B4W, and run-length decoding, which converts the compressed string back to the original. You should be able to compress and decompress in one pass through the string.

The str function converts a number into a string, for example, str(12) = '12'. It will be useful in the encoding step.

The int function converts a string into a number, for example, int('12') = 12. For any string s, s[i].isalpha() returns **True** if the character s[i] is an alphabet character, and **False** otherwise. s.isalpha() returns **True** if *all* the characters in s are alphabet characters. The functions int and isalpha will be useful in the decoding step.

Note

1. The ()'s are necessary in Python 3.x, regardless of whether or not the **print** statement is split across two lines. The ()'s are only necessary in Python 2.x in situations where the **print** statement is split across two or more lines.

2

The Best Time to Party

No one looks back on their life and remembers the nights they got plenty of sleep.
—Author unknown

> Programming constructs and algorithmic paradigms covered in this puzzle: Tuples, lists of tuples, nested **for** loops, and floating-point numbers. List slicing. Sorting.

There is a party to celebrate celebrities that you get to attend because you won a ticket at your office lottery. Because of the high demand for tickets, you only get to stay for one hour, but you get to pick which hour because you received a special ticket. You have a schedule that lists exactly when each celebrity is going to attend the party. You want to get as many pictures with celebrities as possible to improve your social standing. This means you wish to go for the hour when you get to hobnob with the *maximum* number of celebrities and get selfies with each of them.

We are given a list of intervals that correspond to when each celebrity comes and goes. Assume that these intervals are [*i*, *j*), where *i* and *j* correspond to hours. That is, the interval is closed on the left side and open on the right side. This means the celebrity will be partying on and through the *i*th hour, but will have left when the *j*th hour begins. So even if you arrive on the dot on the *j*th hour, you will miss this particular celebrity.

Here's an example:

Celebrity	Comes	Goes
Beyoncé	6	7
Taylor	7	9
Brad	10	11
Katy	10	12
Tom	8	10
Drake	9	11
Alicia	6	8

When is the best time to attend the party? That is, which hour should you go?

By looking at each hour and counting celebrities, you probably figured out that going between the hours of 10 and 11 will get you selfies with Brad, Katy, and Drake. You can't do better than three selfies.

Checking Time and Time Again

The straightforward algorithm looks at each hour and checks how many celebrities are available. A celebrity with [*i*, *j*) is available at hours *i*, *i* + 1, …, *j* − 1. The algorithm simply counts the number of celebrities at each hour and picks the maximum.

Here's the code for this algorithm:

```
1.  sched = [(6, 8), (6, 12), (6, 7), (7, 8),
                (7, 10), (8, 9), (8, 10), (9, 12),
                (9, 10), (10, 11), (10, 12), (11, 12)]

2.  def bestTimeToParty(schedule):
3.      start = schedule[0][0]
4.      end = schedule[0][1]
5.      for c in schedule:
6.          start = min(c[0], start)
7.          end = max(c[1], end)
8.      count = celebrityDensity(schedule, start, end)
9.      maxcount = 0
10.     for i in range(start, end + 1):
11.         if count[i] > maxcount:
12.             maxcount = count[i]
13.             time = i
14.     print ('Best time to attend the party is at',
```

```
        time, 'o\'clock', ':', maxcount,
        'celebrities will be attending!')
```

The input to the algorithm is a schedule, which is a list of intervals. Each interval is a 2-tuple where both entries in each tuple are numbers. The first is the start time and the second is the end time. The algorithm can't modify the intervals, and so we use a tuple to represent them.

Lines 3–7 determine the earliest time that any celebrity will be at the party, and the last time that any celebrity will be. Lines 3 and 4 assume that schedule has at least one tuple in it and initialize the start and end variables. We want start to be the earliest start time of any celebrity and end to be the latest end time of any celebrity. schedule[0] gives the first tuple in schedule. Accessing the two entries of the tuple is exactly the same as accessing a list. Since the tuple is schedule[0], we need another [0] for the first entry to the tuple (line 3) and [1] for the second entry (line 4).

In the **for** loop, we go over all the tuples, naming each one of them c. Note that if we tried to modify c[0] to be 10 (for example), the program would throw an error since c is a tuple. On the other hand, if we had declared sched = [[6, 8], [6, 12], …], we would be able to modify the 6 to 10 (for example), since each element of sched is a list.

Line 8 invokes a function that fills in a list count that will contain, for each time between start and end inclusive, the count of celebrities available at that time.

Lines 9–13 find the time with the maximum number of celebrities by looping through the various times from start to end and keeping track of the maximum celebrity count, namely, maxcount. These lines can be replaced by:

```
 9a.            maxcount = max(count[start:end + 1])
10a.            time = count.index(maxcount)
```

Python provides a function **max** to find the maximum element of a list. In addition, we can use slicing to select a particular contiguous range of elements within the list. In line 9a, we find the maximum element between the indices start and end, inclusive. If we have b = a[0:3], this means that the first three elements of a, namely a[0], a[1], and a[2], are copied into list b, which will have length 3. Line 10a determines the index at which this maximum element is found.

Now for the core of the algorithm, implemented in the function celebrityDensity:

```
1.  def celebrityDensity(sched, start, end):
2.          count = [0] * (end + 1)
```

```
3.      for i in range(start, end + 1):
4.          count[i] = 0
5.          for c in sched:
6.              if c[0] <= i and c[1] > i:
7.                  count[i] += 1
8.      return count
```

The function contains a doubly nested loop. The outer loop iterates through different times, incrementing the time `i` by `1` unit after each iteration, beginning with the time `start` as given by the first argument to **`range`**. For each time, the inner loop (lines 5–7) iterates through the celebrities, and line 6 checks if the celebrity is available at that time. As described earlier, the time needs to be greater than or equal to the celebrity's start time and less than the celebrity's end time.

If we run:

```
bestTimeToParty(sched)
```

the code will print:

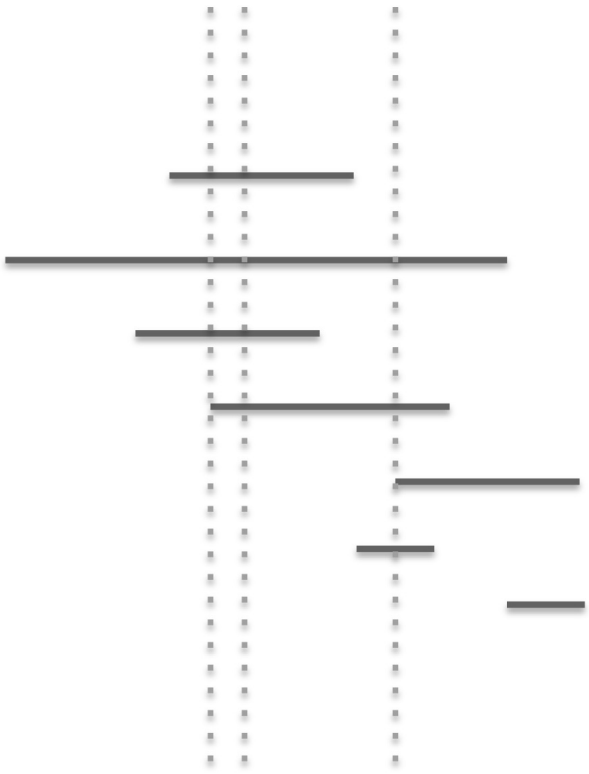
```
Best time to attend the party is at 9 o'clock : 5 celebrities will be attending!
```

This algorithm seems reasonable but is unsatisfying for one primary reason. What is the unit of time? In our example, we could pretend that 6 stands for 6 p.m., 11 for 11 p.m., and 12 for 12 a.m. This means the unit of time is 1 hour. What if celebrities come and go at arbitrary times, as they are wont to do? For example, suppose Beyoncé comes at 6:31 and leaves at 6:42, and Taylor comes at 6:55 and leaves at 7:14. We could make the unit of time 1 minute as opposed to 1 hour. This means performing 60 times as many checks in the loop on line 16. And we would need to look at every second if Beyoncé chose to come at 6:31:15. Celebrities may want to time their entry and exit to the millisecond! (Okay, this is admittedly pretty hard to do even for Beyoncé.) Having to choose a unit of time is annoying.

Can you think of an algorithm that doesn't depend on the granularity of time? The algorithm should use a number of operations that only depends on the number of celebrities and not their schedule.

Being Smart About Checking Times

Let's pictorially represent all celebrities' intervals, with time as the horizontal axis. Here's a possible celebrity schedule:



This picture is evocative—it tells us that if we hold a “ruler” (as shown in dotted lines) at a particular time and count the celebrities whose intervals intersect the ruler, we will know how many we can meet at that time. But we already knew that and coded it in the straightforward algorithm. However, we can make two additional observations from this picture:

. We only need concern ourselves with the start and end points of celebrity intervals because those are the only times that the number of available celebrities changes. There is no reason to compute the number of celebrities available at the second dotted line’s time—it is the same as the first one because no new celebrity has come or gone between the first and second lines/times. (Remember that the fourth celebrity—fourth line from top—has already arrived at the time corresponding to the first dotted line.)
. We can move the ruler from left to right and find the time with the maximum number of celebrities through *incremental* computation, as we will elaborate below.

We keep a count of celebrities, initially zero. Every time we see the start time of a celebrity interval, we increment this count, and every time we see the end time of a celebrity interval, we decrement this count. We also keep track of the

maximum celebrity count. While the count of celebrities changes at the start and end times of celebrity intervals, the maximum count only changes when we see the start point of a celebrity interval.

One crucial point—we have to perform this computation in increasing time to simulate the ruler moving left to right. This means that we have to sort the start times and end times of the celebrities. What we want in the picture above is to see the second-from-top celebrity's start time, followed by the third-from-top celebrity's start time, followed by the first-from-top celebrity's start time, and so on. We'll worry about how to sort these times a bit later, but we are now ready to look at code that corresponds to a more efficient and elegant way of discovering the best time to party.

```
1.  sched2 = [(6.0, 8.0), (6.5, 12.0), (6.5, 7.0),
                (7.0, 8.0), (7.5, 10.0), (8.0, 9.0),
                (8.0, 10.0), (9.0, 12.0), (9.5, 10.0),
                (10.0, 11.0), (10.0, 12.0), (11.0, 12.0)]
2.  def bestTimeToPartySmart(schedule):
3.      times = []
4.      for c in schedule:
5.          times.append((c[0], 'start'))
6.          times.append((c[1], 'end'))
7.      sortList(times)
8.      maxcount, time = chooseTime(times)
9.      print ('Best time to attend the party is at',
                time, 'o\'clock', ':', maxcount,
                'celebrities will be attending!')
```

Note that schedule and sched2 are lists of 2-tuples, as before, where the first number of each tuple is the start time, and the second number of each tuple is the end time. However, we are using floating-point numbers in sched2 as opposed to integers as in schedule to represent time. The numbers 6.0, 8.0, and so on are floating-point numbers. In this puzzle, we are only going to compare these numbers and won't have to perform other arithmetic on them.

On the other hand, the list times, initialized to empty on line 3, is a list of 2-tuples where the first number of each tuple is a time, and the second item is a string label indicating whether the time is a start time or an end time.

Lines 3–6 collect all the start and end times for celebrities, labeling each time as such. This list is not sorted since we cannot assume that the argument schedule is sorted in any fashion.

Line 7 sorts the list times by invoking a sort procedure that we will describe later. Once the list has been sorted, line 8 invokes the critical procedure

chooseTime, which performs the incremental computation of determining celebrity count (or density) at each time.

This code will print the same thing for the original schedule sched and will print for sched2:

```
Best time to attend the party is at 9.5 o'clock : 5 celebrities will be attending!
```

What about sorting the times? We have a list of intervals that we need to convert into times labeled with 'start' or 'end'. And then we just sort the times in ascending order, keeping the labels attached to the times. Here's code that does that:

```
1.  def sortList(tlist):
2.      for ind in range(len(tlist)-1):
3.          iSm = ind
4.          for i in range(ind, len(tlist)):
5.              if tlist[iSm][0] > tlist[i][0]:
6.                  iSm = i
7.          tlist[ind], tlist[iSm] = tlist[iSm], tlist[ind]
```

How does this code work? It corresponds to the simplest possible sorting algorithm,[1] called selection sort. The algorithm finds the minimum time and puts it at the front of the list after the first iteration of the outer **for** loop (lines 2–7). This search of the minimum needs to happen **len**(tlist)-1 times, not **len**(tlist) times—we don't have to find the minimum when there is only one element left.

Finding the minimum element requires going through all the elements in the list, and this happens in the inner **for** loop (lines 4–6). Since there is already an element at the front of the list that needs to remain somewhere in the list, the algorithm swaps the two elements on line 7. Think of line 7 as two assignments that happen in parallel: tlist[ind] gets the old value of tlist[iSm], and tlist[iSm] gets the old value of tlist[ind].

In the second iteration of the outer **for** loop, the algorithm looks at the rest of the list (not including the first item), finds the minimum, and places it as the second item right next to the first, by swapping the element that was in the second position with the minimum in this iteration. Notice that line 4 has two arguments to **range**, to make sure that on each iteration of the outer loop, the inner loop starts at ind, where elements at indices less than ind are already positioned properly. This continues until the entire list is sorted. Since each element of the argument list is a 2-tuple, we have to use the time value in the comparison on line 5, which is the first item in the 2-tuple. This is the reason for

the extra [0]'s on line 5. Of course, we are sorting the 2-tuples; the 2-tuples are being swapped on line 7, and the 'start' and 'end' labels remain attached to the same times.

Once the list has been sorted, procedure chooseTime (shown below) determines the best time and the celebrity count at that time by going through the list *once*.

```
 1.  def chooseTime(times):
 2.      rcount = 0
 3.      maxcount = time = 0
 4.      for t in times:
 5.          if t[1] == 'start':
 6.              rcount = rcount + 1
 7.          elif t[1] == 'end':
 8.              rcount = rcount - 1
 9.          if rcount > maxcount:
10.              maxcount = rcount
11.              time = t[0]
12.      return maxcount, time
```

The number of iterations is twice the number of celebrities, because the list times has two entries (each of which is a 2-tuple) corresponding to the start and end times for each celebrity. Compare this to the doubly nested loop in the straightforward algorithm, which has a number of iterations equaling the number of celebrities times the number of hours (or minutes or seconds as the case may be).

Note that the best time to attend the party will always correspond to the start time of some celebrity. This is because rcount is incremented only at these start times, and therefore the peak is reached at one of these times. We'll exploit this observation in exercise 2.

Sorted Representation

The technique of sorting a list of elements for more efficient processing is a fundamental one. Suppose you have two lists of words and you want to check whether the lists are equal or not. Let's assume each of the lists has no repeated words and they are equal in size with n words each. The obvious way is to take each word in list 1 and check if it exists in list 2. We have to do n^2 comparisons in the worst case, since the check for each word might take n comparisons before it succeeds or fails.

A better way is to sort each of the two lists in alphabetical order of words. Once they are sorted, we can simply check if the first word of sorted list 1 is equal to the first word of sorted list 2, the second word of sorted list 1 is equal to the second word of sorted list 2, and so on. This only requires *n* comparisons.

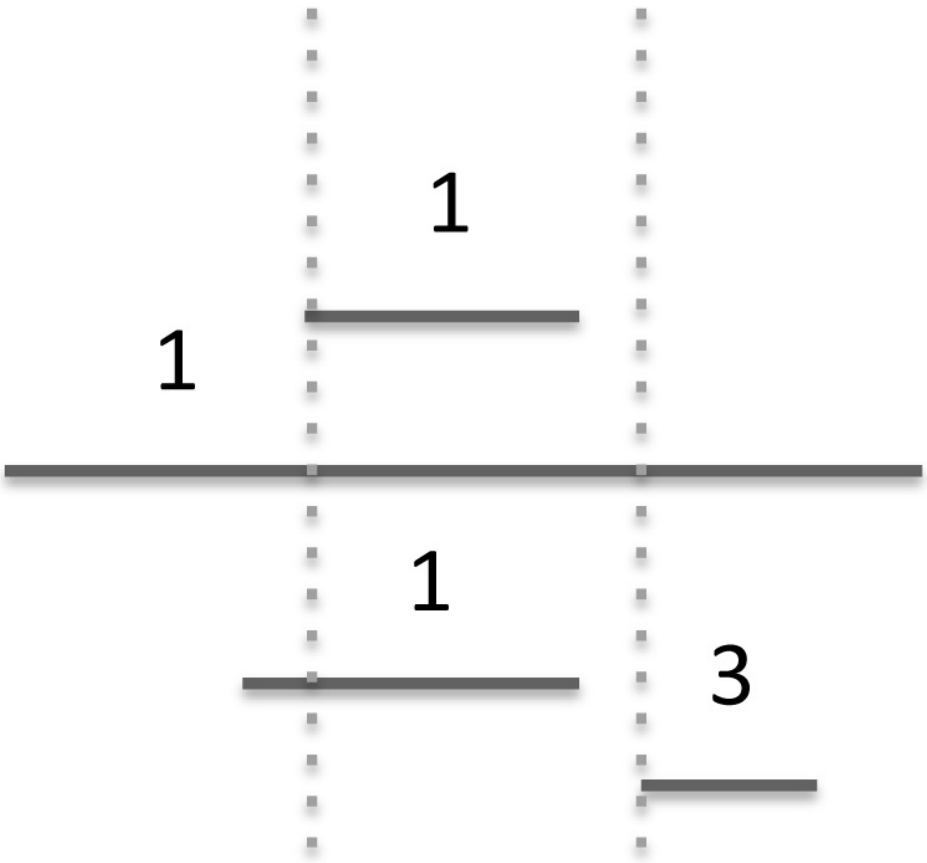
Wait, you say, what about the number of operations required for sorting? Doesn't that selection sort procedure take n^2 comparisons in the worst case? And we are sorting two lists. Stay tuned for better sorting algorithms that only require *n* log *n* comparisons in the worst case, which we will describe later in the book. For large *n*, *n* log *n* is way smaller than n^2, making sorting prior to equality comparison very worthwhile.

Exercises

Exercise 1: Suppose you are yourself a busy celebrity and don't have complete freedom in choosing when you can go to the party. Add arguments to the procedure and modify it so it determines the maximum number of celebrities you can see within a given time range between ystart and yend. As with celebrities, the interval is [ystart, yend), so you are available at all times t such that ystart <= t < yend.

Exercise 2: There is an alternative way of computing the best time to party that does not depend on the granularity of time. We choose each celebrity interval in turn, and determine how many other celebrity intervals contain the chosen celebrity's start time. We pick the time to attend the party to be the *start* time of the celebrity whose start time is contained in the maximum number of other celebrity intervals. Code this algorithm and verify that it produces the same answer as the algorithm based on sorting.

Puzzle Exercise 3: Imagine that there is a weight associated with each celebrity dependent on how much you like that particular celebrity. This can be represented in the schedule as a 3-tuple, for example, (6.0, 8.0, 3). The start time is 6.0, the end time is 8.0, and the weight is 3. Modify the code so you find the time that the celebrities with *maximum total weight* are available. For example, given:



We want to return the time corresponding to the right dotted line even though there are only two celebrities available at that time. This is because the weight associated with those two celebrities is 4, which is greater than the total weight of 3 associated with the three celebrities available during the first dotted line.

Here's a more complex example:

```
sched3 = [(6.0, 8.0, 2), (6.5, 12.0, 1), (6.5, 7.0, 2),
          (7.0, 8.0, 2), (7.5, 10.0, 3), (8.0, 9.0, 2),
          (8.0, 10.0, 1), (9.0, 12.0, 2),
          (9.5, 10.0, 4), (10.0, 11.0, 2),
          (10.0, 12.0, 3), (11.0, 12.0, 7)]
```

For this schedule of celebrities, you want to attend at 11.0 o'clock where the weight of attending celebrities is 13 and maximum.

Note

1. Not necessarily the most efficient, but easiest to code and understand. You will see better sorting algorithms in puzzle 11 and puzzle 13.

3

You Can Read Minds (with a Little Calibration)

Tact is after all a kind of mind reading.
—Sarah Orne Jewett

> Programming constructs and algorithmic paradigms covered in this puzzle: Reading input from a user, and control flow for case analysis. Encoding and decoding information.

You are a magician and a mind reader extraordinaire. Your assistant goes into the audience with an ordinary deck of fifty-two cards while you are outside the room and can't possibly see anything. Five audience members each select one card from the deck. The assistant then gathers up the five cards. The assistant shows the entire audience four cards, one at a time. For each of these four cards, the assistant asks the audience to focus on the card, while you look away and try to read their collective minds. Then, after a few seconds, you are shown the card. This helps you calibrate your mind reading to this particular audience.

After you see these four cards, you claim that you are well calibrated to this audience and leave the room. The assistant shows the fifth card to the audience and puts it away. Again, the audience mentally focuses on the fifth card. You return to the room, concentrate for a short time, and correctly name the secret fifth card.

You are in cahoots with your assistant and have planned and practiced this trick. However, everyone is watching closely and the only information that the

assistant can give you is the four cards.

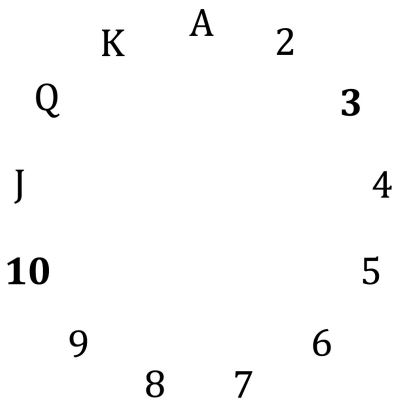
How does this trick work?

It turns out that the order in which the assistant reveals the cards tells the magician what the fifth card is. The assistant needs to be able to decide which of the cards is going to be hidden—he or she cannot allow the audience to pick the hidden card out of the five cards that the audience picks. Here's one way the assistant and the magician can work together.

As a running example, suppose the audience selects: 10♥ 9♦ 3♥ Q♠ J♦

The assistant picks out two cards of the same suit. Given five cards, there will definitely be two with the same suit, since we only have four suits. In the example, the assistant might choose 3♥ and 10♥.[1]

The assistant locates the values of these two cards on the cycle of cards shown below:



For any two distinct values on this cycle, one is always between 1 and 6 hops clockwise from the other. For example, although 10♥ is 7 hops clockwise from 3♥, 3♥ is 6 hops clockwise from 10♥.

One of these two cards is revealed first, and the other becomes the secret card. The card that is revealed is the card *from which* we can reach the other card in 6 or fewer hops clockwise. Thus, in our example, 10♥ would be revealed, and 3♥ would be the secret card, since we can reach 3♥ from 10♥ in 6 hops. (If on the other hand the two cards were 4♥ and 10♥, 4♥ would be revealed since 10♥ is 6 hops clockwise from 4♥.)

- The suit of the secret card is the same as the suit of the first card revealed.
- The value of the secret card is between 1 and 6 hops clockwise from the value of the first card revealed.

All that remains is to communicate a number between 1 and 6. The magician and assistant agree beforehand on an ordering of all the cards in the deck from smallest to largest, such as:

A♣ A♦ A♥ A♠ 2♣ 2♦ 2♥ 2♠ … Q♣ Q♦ Q♥ Q♠ K♣ K♦ K♥ K♠

The order in which the last three cards are revealed communicates the number according to the following scheme:

(small, medium, large) = 1
(small, large, medium) = 2
(medium, small, large) = 3
(medium, large, small) = 4
(large, small, medium) = 5
(large, medium, small) = 6

In the example, the assistant wants to convey the number 6, and so reveals the remaining three cards in large, medium, small order. Here is the complete sequence that the magician sees: 10♥ Q♠ J♦ 9♦
The magician starts with the first card, 10♥, and hops 6 values clockwise to reach 3♥, which is the secret card!

Now that we have a mind-reading algorithm, we are ready to write two programs corresponding to the tasks of the assistant and the magician.

Coding the Assistant's Job

The first program takes five cards as input from the assistant, selects the card that is to be hidden, encodes the remaining four, and prints them out in the correct order. The assistant can read them out to you, the magician, and ask you to guess the hidden card.

```
1.  deck = ['A_C','A_D','A_H','A_S','2_C','2_D','2_H','2_S',
            '3_C','3_D','3_H','3_S','4_C','4_D','4_H','4_S',
            '5_C','5_D','5_H','5_S','6_C','6_D','6_H','6_S',
            '7_C','7_D','7_H','7_S','8_C','8_D','8_H','8_S',
            '9_C','9_D','9_H','9_S','10_C','10_D','10_H',
            '10_S', 'J_C','J_D','J_H','J_S','Q_C','Q_D',
            'Q_H','Q_S', 'K_C','K_D','K_H','K_S']
```

```
2.    def AssistantOrdersCards():
3.        print ('Cards are character strings as shown below.')
4.        print ('Ordering is:', deck)
5.        cards, cind, cardsuits, cnumbers = [], [], [], []
6.        numsuits = [0, 0, 0, 0]
7.        for i in range(5):
8.            print ('Please give card', i+1, end = ' ')
9.            card = input('in above format:')
10.           cards.append(card)
11.           n = deck.index(card)
12.           cind.append(n)
13.           cardsuits.append(n % 4)
14.           cnumbers.append(n // 4)
15.           numsuits[n % 4] += 1
16.           if numsuits[n % 4] > 1:
17.               pairsuit = n % 4
18.       cardh = []
19.       for i in range(5):
20.           if cardsuits[i] == pairsuit:
21.               cardh.append(i)
22.       hidden, other, encode = \
22a.          outputFirstCard(cnumbers, cardh, cards)
23.       remindices = []
24.       for i in range(5):
25.           if i != hidden and i != other:
26.               remindices.append(cind[i])
27.       sortList(remindices)
28.       outputNext3Cards(encode, remindices)
29.       return
```

The list deck on line 1 gives the numeric order of the cards from lowest to highest. Lines 5 and 6 initialize a number of variables. On line 5, you see another example of multiple variables being assigned in a single statement. Lines 7–17 are a **for** loop that asks for keyboard input corresponding to the five cards. The cards have to be entered in exactly the same character format as in deck.

Line 8 is a **print** statement that prints out 'Please give card' and the number of the card. The end = ' ' in the **print** statement prints a space rather than a newline. Line 9 prints out 'in above format:' and takes user character input and writes it into the variable card, which is then appended to the list cards. Together, these two lines produce output that looks like this:

```
Please give card 1 in above format:
```

and wait for a character string from the user. Line 11 is a crucial line that takes the character string corresponding to the card and determines the numeric index associated with it in the list deck. This is important because the index is how we will compare two cards. For example, 'A_C' has index 0, '3_C' has index 8, and 'K_S' has index 51.

Lines 12–17 fill in various data structures required for the encoding task of the assistant. The indices of each of the five cards are stored in cind, and the suits of each of the five cards are stored in cardsuits. The index of each card tells you what the suit is—simply compute the index modulus 4. Each club (suit 0) has an index that is a multiple of 4, each diamond (suit 1) has an index that is a multiple of 4 plus 1, and so on. To obtain the "number" of the card, that is, A (1) through K (13), we simply integer-divide // the index by 4 to get the number, since all the aces are at the beginning of deck, followed by the 2's, and so on. Our algorithm requires the assistant to determine which suit occurs more than once in the five cards—there will be at least one such suit, and if there are two suits occurring twice, we just pick one based on the order of the input cards and this suit number is stored in the variable pairsuit (lines 15–17).

Lines 18–21 determine the two cards that have the same pairsuit. There may be three or more cards with the same suit, but we will only use the first two in the procedure outputFirstCard.

The procedure outputFirstCard (lines 22 and 22a) determines which of these two cards should be hidden, and which should be the first card exposed to the magician. It also determines what number needs to be encoded by the remaining three cards. Notice the \ at the end of line 22—this means lines 22 and 22a should be read together as a single statement. Python will crash if you don't specify this using the \. We will describe outputFirstCard in detail later.

Lines 23–26 remove the hidden card and the first card from the group of five cards, and store their indices in the list remindices, which will have length 3. We sort remindices in increasing order of indices (line 27). Finally, the procedure outputNext3Cards orders the cards based on the number that needs to be encoded—the variable encode stores this number. This procedure will also be described later.

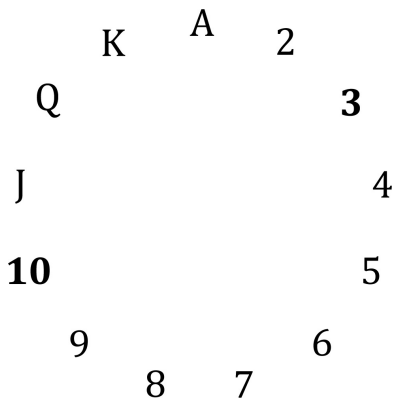
Line 29 has a **return** statement, which signifies the end of the procedure. We have been somewhat cavalier with our procedures thus far and not used **return** statements. Line 29 is not necessary—the procedure will work just fine without it. If you have to return a value, that is, if you are implementing a function, then you definitely need a **return** statement, as shown in the code for our next function

below and functions in prior puzzles.

Here's the function outputFirstCard to determine which of two cards should be hidden. We need to encode a number between 1 and 6 inclusive and choose the two cards appropriately. The function returns the card to be hidden, the first card that will be shown, and the number that needs to be encoded.

```
1.  def outputFirstCard(ns, oneTwo, cards):
2.      encode = (ns[oneTwo[0]] - ns[oneTwo[1]]) % 13
3.      if encode > 0 and encode <= 6:
4.          hidden = oneTwo[0]
5.          other = oneTwo[1]
6.      else:
7.          hidden = oneTwo[1]
8.          other = oneTwo[0]
9.          encode = (ns[oneTwo[1]] - ns[oneTwo[0]]) % 13
10.     print ('First card is:', cards[other])
11.     return hidden, other, encode
```

We are not worried about the suit of the two cards, which is the same. We will use the cycle of cards shown earlier (repeated below) to explain the code.



Suppose the first card is 10♥ and the second is 3♥. Then, line 2 will compute (10 - 3) % 13 = 7. Since encode = 7, we will hide the second card 3♥ and encode (3 - 10) % 13 = 6. In the clockwise direction, incrementing 10 (the first card shown) by 6 gives 3 (the hidden card). On the other hand, if the first card is 3♥ and the second is 10♥, line 2 will compute (3 - 10) % 13 = 6. It will hide the first card 3♥ and encode 6 again.

Here's the procedure outputNext3Cards that orders the three cards stored in the argument list ind to encode the number given by the argument code.

```
1.  def outputNext3Cards(code, ind):
2.      if code == 1:
3.          s, t, f = ind[0], ind[1], ind[2]
4.      elif code == 2:
5.          s, t, f = ind[0], ind[2], ind[1]
6.      elif code == 3:
7.          s, t, f = ind[1], ind[0], ind[2]
8.      elif code == 4:
9.          s, t, f = ind[1], ind[2], ind[0]
10.     elif code == 5:
11.         s, t, f = ind[2], ind[0], ind[1]
12.     else:
13.         s, t, f = ind[2], ind[1], ind[0]
14.     print ('Second card is:', deck[s])
15.     print ('Third card is:', deck[t])
16.     print ('Fourth card is:', deck[f])
```

The procedure assumes that ind[0] < ind[1] < ind[2]. It then changes the order of the cards to encode code.

Finally, here is the sort procedure:

```
1.  def sortList2(tlist):
2.      for ind in range(0, len(tlist)-1):
3.          iSm = ind
4.          for i in range(ind, len(tlist)):
5.              if tlist[iSm] > tlist[i]:
6.                  iSm = i
7.          tlist[ind], tlist[iSm] = tlist[iSm], tlist[ind]
```

This sorting procedure is almost exactly the same as the one from puzzle 2. The only difference is in line 5, where we are comparing the list elements themselves, since the elements are individual cards that can be compared (and not tuples as in puzzle 2).

Coding the Magician's Job

Now that we have automated the assistant's encoding task, we will switch to the magician's job.

Here's a decoding program that takes four cards that are assumed to be correctly encoded and prints the "hidden" card. This is useful for your assistant

to practice with if you are too busy to practice. The assistant selects five cards randomly from the deck, hides a card, and orders them. The assistant then inputs them into the program. The program produces the hidden card, and the assistant can check that the ordering was correctly entered.

```
 1.  def MagicianGuessesCard():
 2.      print ('Cards are character strings as shown below.')
 3.      print ('Ordering is:', deck)
 4.      cards, cind = [], []
 5.      for i in range(4):
 6.          print ('Please give card', i+1, end = ' ')
 7.          card = input('in above format:')
 8.          cards.append(card)
 9.          n = deck.index(card)
10.          cind.append(n)
11.          if i == 0:
12.              suit = n % 4
13.              number = n // 4
14.      if cind[1] < cind[2] and cind[1] < cind[3]:
15.          if cind[2] < cind[3]:
16.              encode = 1
17.          else:
18.              encode = 2
19.      elif ((cind[1] < cind[2] and cind[1] > cind[3])
20.        or (cind[1] > cind[2] and cind[1] < cind[3])):
21.          if cind[2] < cind[3]:
22.              encode = 3
23.          else:
24.              encode = 4
25.      elif cind[1] > cind[2] and cind[1] > cind[3]:
26.          if cind[2] < cind[3]:
27.              encode = 5
28.          else:
29.              encode = 6
30.      hiddennumber = (number + encode) % 13
31.      index = hiddennumber * 4 + suit
32.      print ('Hidden card is:', deck[index])
```

Up until line 10, the two programs are exactly the same, except that on line 5 we only input four cards, rather than five. Lines 11–13 determine the suit of the hidden card and the number of the first card, which is between 1 and 13 inclusive. The rest of the program is about using the numeric order of cards two,

three, and four to determine the difference between the first card and the hidden card. Lines 14 and 15 check whether the three cards are in ascending order. If so, the difference, encode, is 1. If the first card in this group of three cards (second card overall in the four input cards) is the smallest, and the next two cards are in descending order, encode is 2 (line 18).

Lines 19–20 check that the first card of the triple is the middle card, numerically speaking. If that is the case, the difference, encode, will be either 3 or 4. Notice that lines 19 and 20 correspond to the same statement, because parentheses can be used to indicate statement continuation in Python. We added an extra (after the elif, and the matching) on line 20 ends the elif. This extra pair would not have been necessary if we had used a \. Removing them will result in a syntax error if a \ is not used.

Lines 30 and 31 determine what the hidden card is. We already know the suit, and we use the number of the first card and difference, encode, to determine the hidden card's number. But to determine the character string, we need to determine the index of the hidden card in the list deck.

Mastering the Trick in Solitude

What if you, the magician, don't have anyone to practice with? Here's some code that will help.

```
1.  def ComputerAssistant():
2.      print ('Cards are character strings as shown below.')
3.      print ('Ordering is:', deck)
4.      cards, cind, cardsuits, cnumbers = [], [], [], []
5.      numsuits = [0, 0, 0, 0]
6.      number = int(input('Please give random number of' +
                                        ' at least 6 digits:'))
7.      for i in range(5):
8.          number = number * (i + 1) // (i + 2)
9.          n = number % 52
10.         cards.append(deck[n])
11.         cind.append(n)
12.         cardsuits.append(n % 4)
13.         cnumbers.append(n // 4)
14.         numsuits[n % 4] += 1
15.         if numsuits[n % 4] > 1:
16.             pairsuit = n % 4
17.     cardh = []
```

```
18.             for i in range(5):
19.                 if cardsuits[i] == pairsuit:
20.                     cardh.append(i)
21.             hidden, other, encode = \
21a.                     outputFirstCard(cnumbers, cardh, cards)
22.             remindices = []
23.             for i in range(5):
24.                 if i != hidden and i != other:
25.                     remindices.append(cind[i])
26.             sortList(remindices)
27.             outputNext3Cards(encode, remindices)
28.             guess = input('What is the hidden card?')
29.             if guess == cards[hidden]:
30.                 print ('You are a Mind Reader Extraordinaire!')
31.             else:
32.                 print ('Sorry, not impressed!')
```

This program generates five cards “randomly” given a six-digit number as input. The six-digit number that is input is scrambled to produce a set of five cards so it is hard to tell what cards will be produced given a number. You can think of this number as the seed for randomness. The program works as before—it “hides” a card and prints the remaining four in the correct order. Then, it takes the magician’s guess and tells the magician if the guess is correct. This way, a young magician trying to perfect this trick can practice in solitude. To practice properly, input a different number of six or more digits each time.

Lines 6–9 are the main change from AssistantOrdersCards. Notice how the statement that begins on line 6 is across two lines since **input** is similar to **print**. **input** only takes one argument, so we need a + between the two strings.

The magician inputs a large number, which is stored in the variable number. We want to generate “random” indices, five in all, from number. We use some arithmetic on line 8 to generate “random” numbers from number. We could have used random number generators available in Python, but our purpose here is simply to ensure that the magician cannot easily predict what the hidden card is from knowing number. The magician will have to discover the hidden card by decoding information provided by the four revealed cards. Let’s assume our magician is motivated to honestly practice!

Lines 28–32 receive the magician’s guess as input and determine whether the guess is correct or not.

Encoding Information

This puzzle dealt with encoding and communicating information. Say you want to communicate secretly with a friend when others are in earshot. You want to send your friend one bit of information regarding whether you are available for dinner or not. If you start a sentence with “Hey, buddy Alex” versus “Hey, Alex buddy,” Alex could figure out your availability, provided you and Alex had shared the code beforehand. For example, “buddy” before “Alex” codes no availability, and vice versa.

In the card trick, we were able to communicate a number between 1 and 6 because there are six permutations of three cards. In general, if we have n cards, they can be permuted in $n!$ ways, where $n!$ is read n factorial and equals n times $n - 1$ times $n - 2$ all the way down to 1. This means we can communicate a number between 1 and $n!$ inclusive by naming a particular permutation, assuming that the sender and receiver have already exchanged information about what number each permutation corresponds to. Even if a listener knows that there might be secret information being transmitted, as long as the listener does not know what permutation corresponds to what number, the listener will not be able to learn the number based on the permutation.

This means that you and your assistant could agree on different orderings of revealed cards or a different global ordering of cards (see exercise 3), and even audience members who know how the trick works won’t be able to correctly guess the hidden card. So if one of them tries to sabotage the trick by shouting out the answer before you, the assistant can reveal the hidden card, which will likely be different from their guess, and embarrass the heckler. Be warned that doing the trick repeatedly and correctly gives away the agreed-upon ordering to astute observers.

Magic Trick with Four Cards

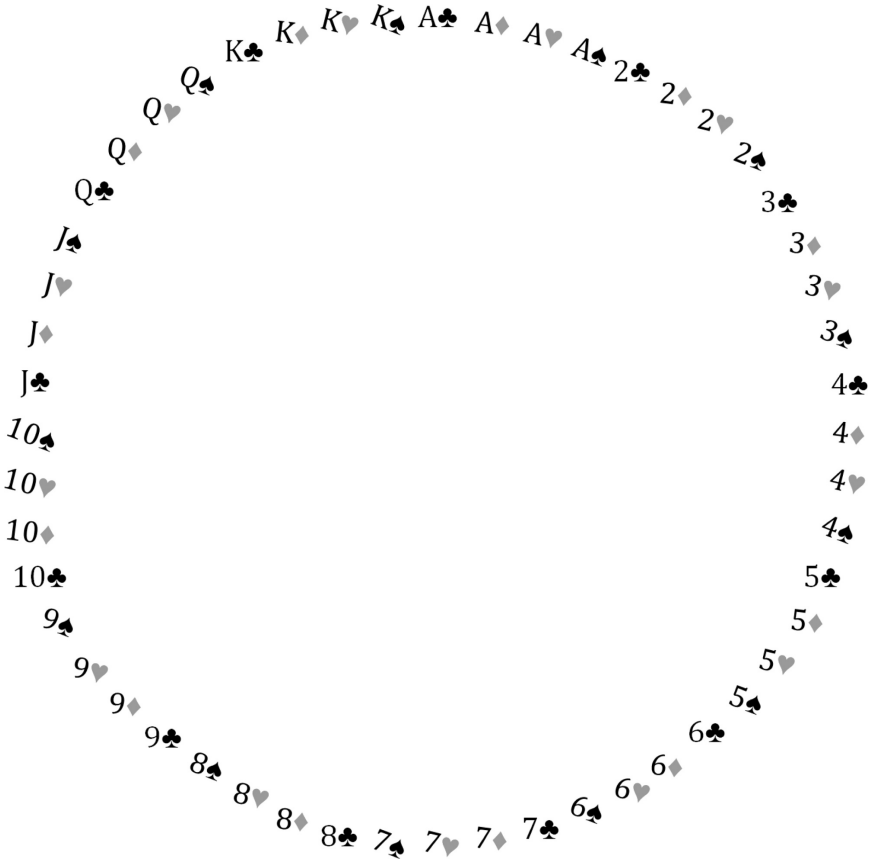
Can you think of a similar trick with only four cards? That is, the assistant asks four people from the audience to each pick a card at random, hides one card, and somehow communicates the hidden card to the magician by announcing the three remaining cards. As before, the assistant gets to choose what card to hide.

This seems difficult, because unlike in the five-card puzzle, the suits of the four cards could all be different. This means the assistant has to communicate the suit of the hidden card in addition to the number of the hidden card, all with only three cards.

There are many ways the assistant can communicate information. The

assistant could both announce the cards theatrically in some order and place the announced cards on a table faceup or facedown without drawing attention. The interesting question is how much information the assistant has to convey given the freedom in being able to pick the hidden card.

Consider the larger cycle of cards shown below, which is similar to the 13-cycle shown earlier but includes the whole deck of fifty-two cards. The order shown is the same as the ordering in the variable deck. The four cards will appear randomly in distinct locations on the circle.



What is the smallest distance between any pair of cards in the *worst* case? In the worst case, the cards are equally spread apart, for instance, Q♣, 2♦, 5♥, and 8♠. In this case each pair of cards will be 13 hops apart. This means that, regardless of the chosen cards, the assistant can *always* choose a pair that is *at most* 13 hops apart. Similar to our five-card trick, the card that is revealed first is the card from which we can reach the other hidden card clockwise in 13 or fewer hops.

Think of the subtlest way an assistant can communicate a number between 1 and 13 to the magician. One way would be to place the hidden card on the table,

facedown of course, and then place each of the three announced cards either to the left or right of the hidden card. Left versus right gives one bit of information to the magician for each card. Placing the three announced cards all faceup or all facedown gives one more bit of information. With four bits you can encode numbers between 0 and 15. As with most magic tricks, distracting the audience is important to this trick!

Exercises

Exercise 1: There is a small bug in the ComputerAssistant program because we got lazy with the following code:

```
7.          for i in range(5):
8.              number = number * (i + 1) // (i + 2)
9.              n = number % 52
```

We are using an input number to generate five "random" cards. The problem with this particular strategy is that it does not check that the five cards obtained are distinct. In fact, if you input the number 888888, the following five cards are generated in this order: ['A_C', 'A_C', '7_H', 'J_D', 'K_S']. You can see the problem here—we are not checking for duplicate cards in ComputerAssistant. Fix this procedure to check for distinct cards, and generate additional "random" numbers by running the loop above for additional iterations till five distinct cards are generated.

Exercise 2: Modify ComputerAssistant so that in the case of two pairs of cards with the same suit, the hidden and first cards are selected such that the number to be encoded is the smallest possible.

Exercise 3: Some magicians prefer a different ordering of the cards, where the suit of the card is the major determinant of the order as opposed to the number, as shown in the following.

```
deck = ['A_C','2_C','3_C','4_C','5_C','6_C','7_C','8_C',
        '9_C','10_C','J_C','Q_C','K_C','A_D','2_D','3_D',
        '4_D','5_D','6_D','7_D','8_D','9_D','10_D','J_D',
        'Q_D','K_D','A_H','2_H','3_H','4_H','5_H','6_H',
        '7_H','8_H','9_H','10_H','J_H','Q_H','K_H','A_S',
        '2_S','3_S','4_S','5_S','6_S','7_S','8_S','9_S',
        '10_S','J_S','Q_S','K_S']
```

Modify ComputerAssistant so the magician can practice with the above order. Note that the computation of numbers and suits of the cards given the index of the

card needs to change. And, of course, the ordering of cards read out by the assistant will change as a result. A few details to get right in this variant solution to the puzzle.

Puzzle Exercise 4: Code ComputerAssistant4Cards, which allows you to practice the four-card trick. You can use the encoding strategy described, where information is provided based on (1) whether each card is to the left or right of the hidden card on the table, and (2) whether all three cards are faceup or all are facedown. Or you can devise your own encoding. If you devise your own encoding, make sure that the trick is guaranteed to work for all possible selections of cards.

Note

1. Mathematically inclined readers may recognize this as the pigeonhole principle. Given n holes and $n + 1$ pigeons that each need to fly through some hole, there is at least one hole through which two pigeons will fly.

4

Keep Those Queens Apart

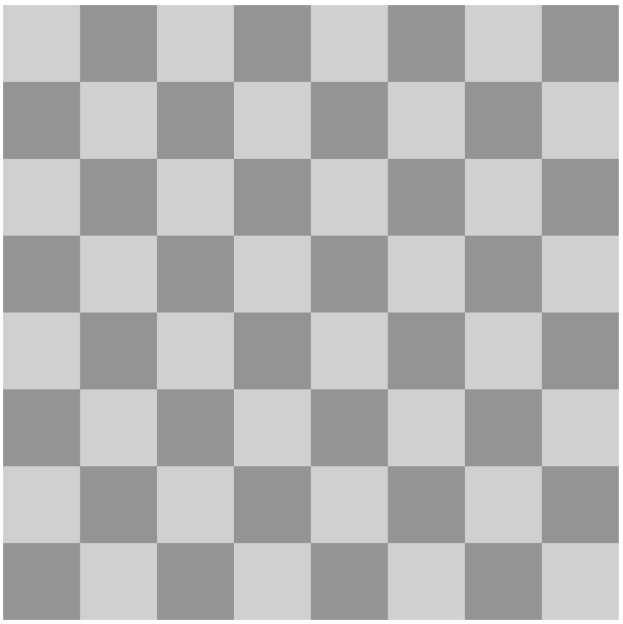
When the world says, “Give up,” Hope whispers, “Try it one more time.”
—Author unknown

Programming constructs and algorithmic paradigms covered in this puzzle: Two-dimensional lists, **while** loops, **continue** statements, and argument defaults in procedures. Exhaustive search via iteration. Conflict detection.

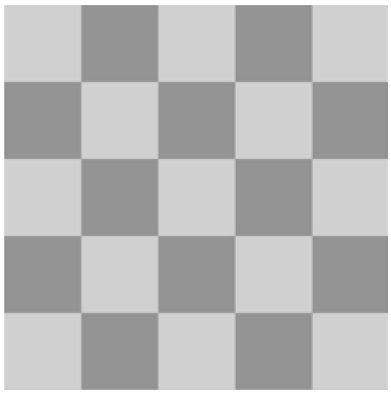
The eight-queens problem on a chessboard means finding a placement of eight queens such that no queen attacks any other queen. This means that:

. No two queens can be on the same column.
. No two queens can be on the same row.
. No two queens can be on the same diagonal.

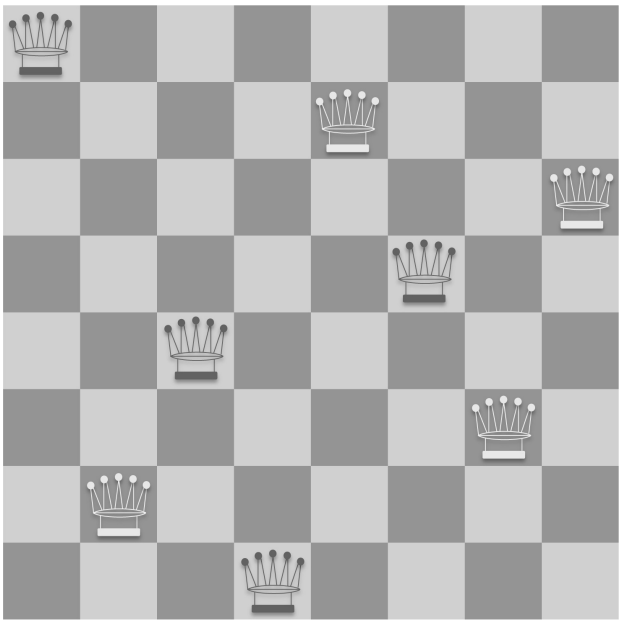
Can you find a solution? (See the following board.)



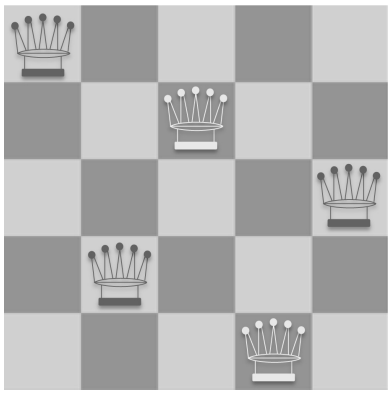
If eight queens are too many to handle, try the five-queens problem below.



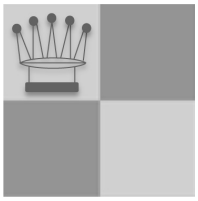
A solution to the eight-queens problem is shown in the following board. This is not the only solution.



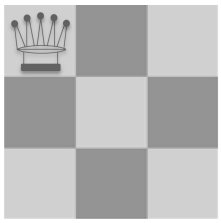
And here’s a solution to the five-queens problem.

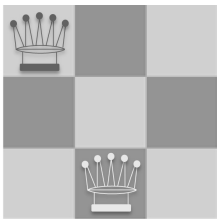


But how can we find the above solutions or different ones? Let’s simplify the problem first. Suppose we look at a smaller, 2 × 2 board (below). Can we place two queens so they don’t attack each other? The answer is no, since a queen on any square of a 2 × 2 board can attack every other square.



What about a 3 × 3 board? Here is an attempt:

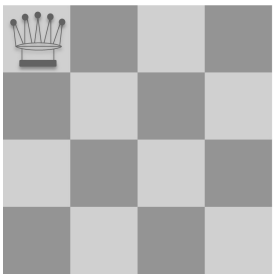


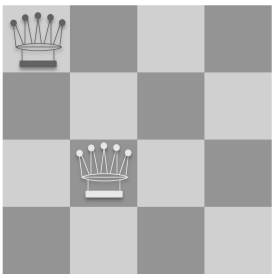


For the first placement (above left), we can see that the queen attacks six other squares and occupies one. So seven squares are disallowed by the very first queen placement, leaving two squares available. When we put the second queen on one of those squares, there are no squares available. Of course, we could have tried putting the first queen in a different spot, but that won't help. The first queen, regardless of where it is placed, will occupy and attack at least seven squares, leaving two. There is no solution for a 3 × 3 board.

Systematic Search

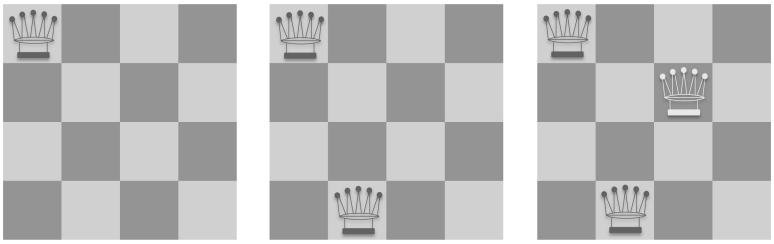
What about a 4 × 4 board? Let's be systematic about searching for a solution. We will try to place queens column by column and change placements if we fail. We start by placing a queen in the top left corner in the first column as shown below (first picture). There are two choices in placing a queen on the second column, and we choose one, as shown in the second picture below. Now we are stuck! We can't place a queen in the third column.



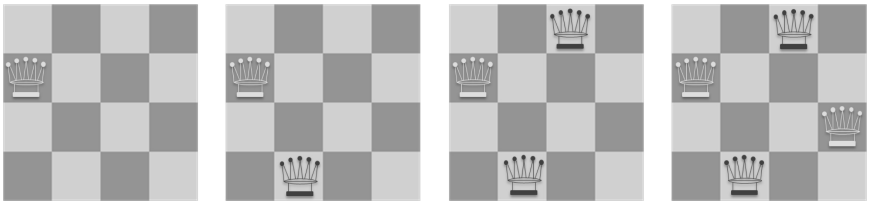


Note that when we tried to place the third queen in the third column above, we simply checked that the third queen didn't conflict with the first and second. We do not need to check that the first two queens do not conflict with each other. This is because we did that check when we placed the second queen. This will be important to remember when we look at the code to check for conflicts.

Given that we failed at finding a solution, should we give up? No, because we could try a different positioning for the second queen. Here we go:



We went further, but are stuck again, this time on the fourth and last column. But we are not out of options. We arbitrarily chose the top corner for the first queen, and we could try a different position. (There are four options for the first queen.) Here we go:



Success! The four-queens problem has a solution.[1]

It will take a normal person quite a long time to use this strategy to find a solution to the eight-queens problem. But such an exhaustive brute-force strategy should work. And computers can do calculations billions of times faster than humans, so if we can code up our strategy, we will be able to run the program and find a solution in a matter of seconds.

The first step in writing code for eight-queens is to decide on a data structure for the problem: How are we going to represent the board and the positions of the queens?

Board as a 2-D List/Array

We have already seen 1-D lists/arrays in Python in puzzle 1, with the `caps` variable. Since a chessboard is a two-dimensional grid, a natural representation is a two-dimensional array:

```
B =   [[0, 0, 1, 0],
       [1, 0, 0, 0],
       [0, 0, 0, 1],
       [0, 1, 0, 0]]
```

Read the array just like you would look at the board, with the `0`'s being blank squares and the `1`'s being queens. `B` is an array of (single-dimensional) arrays: `B[0]` is the first row, `B[1]` is the second row, and so on. So `B[0][0] = 0`, `B[0][1] = 0`, `B[0]`

[2] = 1, and B[0][3] = 0. As a further example, B[2][3] = 1. This is the last 1 on the third row. The variable B above represents the solution that we came up with on the 4 × 4 board shown previously.

We will need to check that there is exactly one B[i][j] that is a 1 for each given i and varying j, and for each given j and varying i. We will also need to check for diagonal attacks. For example, B[0][0] = 1 and B[1][1] = 1 is an invalid configuration (not a solution).

Here's code that checks whether a given 4 × 4 board with queens on it violates the rules. Note that this code does not check that there are four queens on the board. So an empty board will satisfy the checks. It is certainly possible that two queens by themselves will violate the checks. Remember, we want to follow the iterative strategy we showed earlier in placing one queen at a time on the board in a new column and checking for conflicts.

```
1.  def noConflicts(board, current, qindex, n):
2.      for j in range(current):
3.          if board[qindex][j] == 1:
4.              return False
5.      k = 1
6.      while qindex - k >= 0 and current - k >= 0:
7.          if board[qindex - k][current - k] == 1:
8.              return False
9.          k += 1
10.     k = 1
11.     while qindex + k < n and current - k >= 0:
12.         if board[qindex + k][current - k] == 1:
13.             return False
14.         k += 1
15.     return True
```

Given an $N \times N$ board, any positioning of N queens on the board is called a configuration. If there are fewer than N queens, we will call it a partial configuration. Only if a configuration (with N queens) satisfies all three attack rules will we call it a solution.

The procedure noConflicts(board, current) checks a partial configuration to see if it violates the row and the diagonal rules. It takes as an argument qindex, which is the row index of the queen that has been placed on the column current. The procedure assumes that only one queen will be placed in any given column—our FourQueens iterative search procedure below has to guarantee that (and it does).

The value of current can be less than the size of the board; the columns after current are empty. The code checks whether the column numbered current has a queen conflicting with existing queens in columns with numbers less than current. This is all we want, since we want to mimic the manual iterative procedure we described earlier. When current = 3, for example, in a call to noConflicts, we are not checking that board[0][0] and board[0][1] are both 1's (row attack across the first two columns) or that board[0][0] and board[1][1] are both 1's (diagonal attack across the first two rows and columns). We can get away with this provided we call noConflicts *each* time we add to the partial configuration by positioning a new queen.

Lines 2–4 check that there is not already a queen in row qindex. Lines 5–9 and 10–14 check for the two forms of diagonal attacks. Lines 5–9 check for ↗ attacks by decrementing qindex and current in proportion to an edge of the board. Lines 10–14 check for ↘ attacks by decrementing current and incrementing qindex in proportion till an edge of the board is reached. The board beyond the column current is assumed to be empty, so there is no need to increment current.

Now we are ready to invoke the checking-conflicts procedure while placing queens.

```
1.  def FourQueens(n=4):
2.      board = [[0,0,0,0], [0,0,0,0],
                 [0,0,0,0], [0,0,0,0]]
3.      for i in range(n):
4.          board[i][0] = 1
5.          for j in range(n):
6.              board[j][1] = 1
7.              if noConflicts(board, 1, j, n):
8.                  for k in range(n):
9.                      board[k][2] = 1
10.                     if noConflicts(board, 2, k, n):
11.                         for m in range(n):
12.                             board[m][3] = 1
13.                             if noConflicts(board, 3, m, n):
14.                                 print (board)
15.                     board[m][3] = 0
16.                 board[k][2] = 0
17.             board[j][1] = 0
18.         board[i][0] = 0
19.     return
```

Thanks to lines 4 and 18, lines 6 and 17, lines 9 and 16, and lines 12 and 15, we are guaranteed that there is exactly one queen in each column—this is an invariant enforced by FourQueens. Initially the board B is empty and each pair of lines places one queen and removes it. This is why noConflicts does not need to check for column attacks, and only checks for row and diagonal attacks between the newly placed queen and existing queens.

Line 4 places the first queen on the board, and there cannot be any conflicts with only one queen, hence we do not have to call noConflicts after this placement. For the second and subsequent queen placements (lines 6, 9, and 12) we do have to check for conflicts.

While we have written noConflicts for general n, FourQueens assumes that n = 4, so we hardcoded the argument n = 4 when invoking it. We could easily have replaced all occurrences of n in FourQueens with the number 4, but we chose this description to emphasize that noConflicts is general enough to be invoked with arbitrary n, but FourQueens, as its name gives away, is not.

If you run FourQueens, you get:

```
[[0, 0, 1, 0],
 [1, 0, 0, 0],
 [0, 0, 0, 1],
 [0, 1, 0, 0]]
[[0, 1, 0, 0],
 [0, 0, 0, 1],
 [1, 0, 0, 0],
 [0, 0, 1, 0]]
```

Each row of the board is listed, from top to bottom. The code produces two solutions, each on four lines above, the first of which is the solution we manually discovered earlier. We stopped after we found the first solution, but if we had kept going, we would have discovered the second.

To code EightQueens we can simply add more loops to FourQueens. (As if four nested loops aren’t enough!) Before we do that, we will look at a better data structure for partial or full configurations that is not only more compact, but also makes the checks for the three rules above easier.

Board as a 1-D List/Array

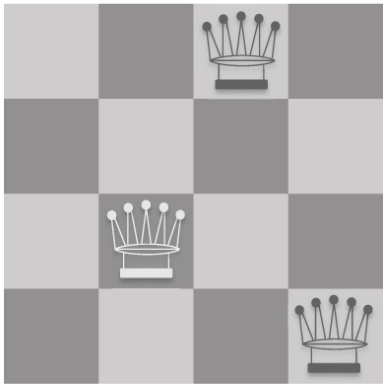
The 2-D list representation of the board was natural and it clearly served our purpose. It turns out that, because we are only looking for solutions where there

is a single queen on each column (or row, for that matter), we can get away with using a one-dimensional array where each index represents each column of the board and the entry represents the row on which the queen resides. Consider the array of size 4:

a	*b*	*c*	*d*

The values *a*, *b*, *c*, *d* can vary from −1 to 3. Here, −1 means there is no queen in the corresponding column, 0 means there is a queen in the first row of that column, and 3 means there is a queen in the last row. Here's a general example that should make things clear:

–1	2	0	3



This data structure can represent partial configurations in the sense that there are only three queens on the 4 × 4 board above. This is important because we will want to build up a solution from an empty board as in our first algorithm. Of course, in our algorithm we placed queens in columns beginning from left to right, but that was an arbitrary choice.

We can now write code to check for the three rules, given our new representation. Notice that we get the first rule for free, in the sense that we cannot place two queens on the same column, since we can only have a single number between −1 and $n - 1$ as the value stored at each array index. Not only is the 1-D representation more compact, it also saves us from checking one of the rules. For the second rule, which checks that no row contains two or more queens, we just need to make sure that no number (other than −1) appears more than once in the array (lines 3–4 below). The third rule involves slightly more computation (lines 5–6 below).

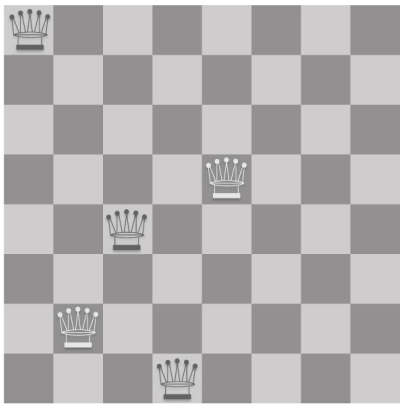
```
1.  def noConflicts(board, current):
```

```
2.          for i in range(current):
3.              if (board[i] == board[current]):
4.                  return False
5.              if (current - i == abs(board[current] - board[i])):
6.                  return False
7.          return True
```

Lines 3–4 check that there is no row attack. They assume that board[current] has a nonnegative value enforced by the calling procedure EightQueens. If that value equals the value in any preceding column, we have an invalid configuration.

Line 5 checks for diagonal attacks—this is much simpler to do in this representation. Why the **abs** in line 5? We have to check for two diagonals corresponding to the ↖↘ and ↙↗ directions. Look at the example below:

0	6	4	7	3	–1	–1	–1



The variable current = 4 and we are trying to place a queen in the column numbered 4. board[1] = 6 and board[4] = 3, so the diagonal check will fail for i = 1 on line 5.

```
{current = 4} - {i = 1} == abs({board[current] = 3} - {board[i] = 6})
```

Remember that we had to do the two diagonal checks in our old representation as well.

Here's EightQueens using our new, compact representation.

```
1.  def EightQueens(n=8):
2.      board = [-1] * n
3.      for i in range(n):
4.          board[0] = i
5.          for j in range(n):
6.              board[1] = j
7.                  if not noConflicts(board, 1):
```

```
                    continue
                for k in range(n):
                    board[2] = k
                    if not noConflicts(board, 2):
                        continue
                    for l in range(n):
                        board[3] = l
                        if not noConflicts(board, 3):
                            continue
                        for m in range(n):
                            board[4] = m
                            if not noConflicts(board, 4):
                                continue
                            for o in range(n):
                                board[5] = o
                                if not noConflicts(board, 5):
                                    continue
                                for p in range(n):
                                    board[6] = p
                                    if not noConflicts(board, 6):
                                        continue
                                    for q in range(n):
                                        board[7] = q
                                        if noConflicts(board, 7):
                                            print (board)
    return
```

We place a queen simply by assigning a nonnegative number to the column that we are working on. There is no need to reset the value to `0` as we had to do in the old algorithm, because changing the number effectively removes a queen from the old position and moves it to a new one. That is, if `board[0] = 1`, and we change it to `board[0] = 2`, we just moved the queen. Again, the more compact representation saves us from writing code.

Remember that we have to check for conflicts each time we place a queen against the preceding columns. This is why we have a `noConflicts` call each time a queen is placed. To avoid having to indent the code even more than we already have to with eight loops, we employ the **`continue`** statement. If there is a conflict, that is, the **`if`** statement's predicate returns **`False`**, we jump to the next loop iteration without executing any more statements.

```
    for j in range(n):
```

```
6.              board[1] = j
7.              if not noConflicts(board, 1):
8.                  continue
9.              for k in range(n):
```

If line 7 above produces **False** for the call to noConflicts, we will jump back to line 6 with an incremented value of j without bothering with line 9 or subsequent statements. This avoids having to enclose the **for** loop beginning on line 9 with the **if** statement of line 7 (without the **not**) as we did in the FourQueens code.

If we run the EightQueens code, it prints out different solutions. Here is a sample:

```
[0, 4, 7, 5, 2, 6, 1, 3]
[0, 5, 7, 2, 6, 3, 1, 4]
[0, 6, 3, 5, 7, 1, 4, 2]
[0, 6, 4, 7, 1, 3, 5, 2]
```

The last one is the solution we showed you at the beginning of this puzzle description. If you consider rotations and mirrored solutions as the same solution, there are twelve distinct solutions.

If we follow **print** (board) on line 32 with a **return**, we will only get the first solution:

```
29.                             for q in range(n):
30.                                 board[7] = q
31.                                 if noConflicts(board, 7):
32.                                     print (board)
33.                                     return
34.         return
```

The eight-queens iterative code is the ugliest code in the entire book! Imagine what would happen if you wanted to solve the fifteen-queens problem. Fortunately, we'll show you elegant *recursive* code that solves the *N*-queens problem for arbitrary *N* in puzzle 10.

Iterative Enumeration

The main algorithmic paradigm embodied by our *N*-queens code is that of iterative enumeration. We go column by column, and within each column, we go row by row. To ensure that we find a solution we need the enumeration to be *exhaustive*—if we skip placing a queen in a column or row, we may miss finding a solution. By numbering columns and rows and iterating through each of them,

we ensure an exhaustive search.

The other algorithmic paradigm illustrated in the code is conflict detection during the iterative search. In the four-queens problem, for example, we could have placed four queens on the board, one in each column, and only *then* checked for conflicts. This is illustrated in the code below:

```
1.      for i in range(n):
2.          board[i][0] = 1
3.          for j in range(n):
4.              board[j][1] = 1
5.              for k in range(n):
6.                  board[k][2] = 1
7.                  for m in range(n):
8.                      board[m][3] = 1
9.                      if noConflictsFull(board, n):
10.                         print (board)
11.                     board[m][3] = 0
12.                 board[k][2] = 0
13.             board[j][1] = 0
14.         board[i][0] = 0
15.     return
```

This code is strictly worse than what we showed you in terms of performance and code complexity. It is worse in terms of performance because we will create configurations where there are three queens on the same row, for example. It is worse in terms of complexity because we need a more complex noConflictsFull check (line 9) rather than the incremental check we do in noConflicts where we simply check if the most recently placed queen conflicts with the already placed queens. In noConflictsFull, we have to check that each row has exactly one queen in it, and that each pair of queens does not have either type of diagonal conflict. There is repeated work done for configurations that are close to each other across the 4^4 calls to noConflictsFull.

We won't bother with implementing noConflictsFull. The main purpose in showing you the above code is so you appreciate the FourQueens code a little more.

Exercises

Exercise 1: Modify the EightQueens code so it takes as an additional argument —the number of solutions you want to find—and prints that many, assuming

that many exist. Note that default arguments have to be after non-default arguments, so the new argument has to be the first one, followed by the `n=8` default argument.

Puzzle Exercise 2: Modify the `EightQueens` code so it looks for solutions with a queen already placed in a list of locations. You can use a 1-D list `location` as an argument that has nonnegative entries for certain columns that correspond to fixed queen positions. For example, `location = [-1, 4, -1, -1, -1, -1, -1, 0]` has two queens placed in the second and eighth columns. Your code should produce `[2, 4, 1, 7, 5, 3, 6, 0]` as a solution consistent with the prescribed queen locations.

Exercise 3: Reduce the level of indentation in the `FourQueens` code by using the **`continue`** statement. Both solutions should be printed. This is a little trickier than you might think!

Note

1. In fact it has two. Try to find a different one.

5

Please Do Break the Crystal

Tell the broken plate you are sorry.
—Mary Robertson

> Programming constructs and algorithmic paradigms covered in this puzzle: Break statements, radix representations.

You are tasked with determining the "hardness coefficient" of a set of *identical* crystal balls. The famous Shanghai Tower, completed in 2015, has 128 floors,[1] and you have to figure out from how high you can drop one of these balls so it doesn't break, but rather bounces off the ground below. We will assume that the surrounding area has been evacuated while you conduct this important experiment.

What you would like to do is report to your boss the highest floor number of Shanghai Tower from which the ball does *not* break when dropped. This means that if you report floor *f*, the ball does not break at floor *f*, but does break at floor $f + 1$ (or else you would have reported $f + 1$). Your bonus depends on how high a floor you report, and if you report a floor *f* from which the ball breaks, you face a stiff fine, which you want to avoid at all costs.

Once a ball breaks, you can't reuse it, but you can if it survives. Since the ball's velocity as it hits the ground is the sole factor determining whether it breaks, and this velocity increases[2] with each floor number, you can assume that if a ball does not break when dropped from floor *x*, it will not break from any

floor number < *x*. Similarly, if it breaks when dropped from floor *y*, it will break when dropped from any floor number > *y*.

Sadly, you are not allowed to take an elevator because the shiny round objects you are carrying may scare off other passengers. You would therefore like to minimize the number of times you drop a ball, since it is a lot of work to keep climbing up stairs.

Of course, the big question is how many balls you are given. Suppose you are given exactly one ball. You don't have much freedom to operate. If you drop the ball from floor 43 (for instance) and it breaks, you don't dare report floor 42, because it might break when dropped from floor 42, floor 41, or floor 1 for that matter. You will have to report floor 1, which means no bonus. With one ball, you will have to start with floor 1—if the ball breaks, report floor 0, and if not you move up to floor 2, all the way till floor 128. If it doesn't break at floor 128, you happily report 128. If the ball breaks when dropped from floor *f*, you will have dropped the ball *f* times. The number of drops could be as large as 128, floors 1 through 128 inclusive.

What if you have two balls? Suppose you drop one ball from floor 128. If it does not break, you report floor 128 and you are done with the experiment, and rich. However, if it breaks, you are down to one ball, and all you know is that the balls you are given definitely break at floor 128. To avoid a fine and to maximize your bonus, you will have start with the second ball at floor 1 and move up as described earlier, possibly all the way up to floor 127. The number of drops in the worst case is 1 drop (from floor 128) plus drops from floors 1 through 127 inclusive, a total of 128. No improvement from the case of one ball.

Intuition says you should guess the midpoint of the interval [1, 128] for the 128-floor building. Suppose you drop a ball at floor 64. There are two cases, as always:

. The ball breaks. This means you can focus on floors 1 through 63—that is, interval [1, 63]—with the remaining ball.
. The ball does not break. This means you can focus on floors 65 through 128—that is, interval [65, 128]—with both balls.

The worst-case number of drops is 64, because in case 1 you will need to start with the lowest floor in the interval and work your way up. Better than 128, but only by a factor of two.

You would like to do better than the worst case of 64 drops when you have two balls. You don't want to give up any part of your bonus, and a fine is a no-

no.

Can you think of a way to maximize your bonus and avoid a fine while using no more than 21 drops in the case of two balls? What if you had more balls, or what if Shanghai Tower suddenly doubled in height (floor count)?

Efficient Search with Two Balls

We should be able to do better than 64 drops. The problem with beginning with floor 64 when we only have two balls is that if the first ball breaks, we have to start with floor 1 and go all the way to floor 63. What if we start at floor 20? If the first ball breaks, we have to search the smaller interval [1, 19] with the second ball one floor at a time. That is 20 drops total in the worst case. If the first ball does not break, we search the large interval [21, 128], but we have two balls. Let's next go to floor 40 and drop the first ball (second drop of the first ball). If the first ball breaks we search [21, 39] one floor at a time. This is, in the worst case, a total of 2 drops for the first ball (at floors 20 and 40) and 19 drops for the second ball, for a total of 21. On to floor 60, and so on. Trying floors 20, 40, 60, 80, and so on, is going to get us a worst-case solution of fewer than 30 drops for sure.

Our purpose here is not just to solve a specific 128-floor problem. Is there a general algorithm where, given an *n*-floor building and two balls, we can show a symbolic worst-case bound of func(*n*) where func is some function? Then we can apply this algorithm to our specific 128-floor problem.

The key is to distribute the floors properly. Suppose we use a strategy where we drop the first ball at floors k, $2k$, $3k$, …, $(n/k - 1)k$, $(n/k)k$. Let's assume that the first ball does not break till the last drop. In this case, we have n/k drops of the first ball, and we have to search the interval $[(n/k - 1)k + 1, (n/k)k - 1]$, which is $k - 1$ drops of the second ball in the worst case. That is, a total of $n/k + k - 1$ drops in the worst case.

Therefore, we want to choose k to minimize $n/k + k$. The minimum happens when k is $\sqrt{n}$. (Occasionally, high school calculus comes in handy!) The worst-case number of drops will be $2\sqrt{n} - 1$. For our 128-floor example, we should space the first ball's drops $\sqrt{128} = 11$ floors apart. This will get us to 21 drops worst-case. We have rounded the fractional square root down. We could also round up.

Is 21 drops the best we can do? We did make the assumption that the floors we would drop from were evenly distributed: k, $2k$, $3k$, and so on. It turns out

that distributing the floors carefully and unevenly can shrink the number of drops required below 21, and we will explore uneven distribution later. For now, we will turn our focus to the general setting of having $d \geq 1$ balls and n floors, and see if we can come up with a strategy similar to what we used for one ball and two balls, but which requires fewer drops as the number of balls increases.

Efficient Search with *d* Balls

When we have one ball, that is, $d = 1$, we have no choice but to start at the first floor and work our way upward. When we have two balls, starting at floor $\sqrt{n}$ (or $n^{½}$) is the way to go, as we determined. Should we start at floor $n^{1/d}$ when we have d balls? What happens when a ball breaks?

Let's consider base-r representations of numbers. When $r = 2$, we have the binary representation, when $r = 3$, the ternary representation, and so on. Given the number of floors n, and the number of balls d, choose r such that $r^d > n$. So if $n = 128$, and $d = 2$, we will choose $r = 12$ (this corresponds to rounding up the fractional square root). If $d = 3$, we will choose $r = 6$, since $5^3 < 128$ and $6^3 > 128$. Let's assume for the purposes of our next example that $d = 4$, which means $r = 4$, when $n = 128$.

The numbers we consider are d-digit, base-r numbers. For our chosen $r = 4$, $d = 4$ example, the smallest number is 0000_4 (0 in decimal) and the largest number is 3333_4 (255 in decimal). As a reminder, in a radix-4 representation, a number such as 1233_4 is equivalent to $1 \times 4^3 + 2 \times 4^2 + 3 \times 4^1 + 3 \times 4^0 = 111$ in decimal.

We drop the first ball at floor 1000_4 (floor 64). If it does not break, we move to floor 2000_4 (floor 128) and drop the ball from there. If it still does not break, we are done. If it does break, we are down to three balls, but we know now that we only have to search the range $[1001_4, 1333_4]$, which is [65, 127] in decimal. We will move to the second phase, using the highest floor in the first phase where the ball did not break.

In our second phase, we work with the second ball and the second digit from the left in our base-r representation. Let's assume that in the first phase the ball broke at floor 2000_4, and did not break at floor 1000_4. In the second phase, our first drop will be from floor 1100_4 (floor 80, which is inside the range [65, 127]). In the second phase we keep incrementing the second digit in our base-r representation and drop balls from corresponding floors. We will drop from floor 1100_4, 1200_4, and 1300_4, in that order. As before, we move to the third phase with the highest floor from which the ball did not break in the second phase. For

the purposes of our example, let's assume that is floor 1200_4. We now need to search the interval [1201_4, 1233_4], since the ball broke at floor 1300_4. This corresponds to [97, 111] in decimal.

In phase 3, we drop the third ball from floor 1210_4—we increment the third digit of our representation from the floor we found in the second phase. Floor 1210_4 is followed by 1220_4 and 1230_4. Let's assume the ball breaks when dropped from floor 1230_4. This means we move to the fourth phase with floor 1220_4. The range we are searching is [1221_4, 1223_4], corresponding to [105, 107] in decimal.

In our fourth and last phase, we drop the fourth ball from floors 1221_4, 1222_4, and 1223_4, incrementing the fourth digit in our representation. If the ball does not break, we report floor 1230_4. If the ball breaks when dropped from any of the floors (e.g., 1223_4), we report the floor that is one below (e.g., 1222_4).

What is the maximum number of drops we will make? In each phase, we drop the ball $r - 1$ times at most. Since there are at most d phases, the total number of drops is at most $d \times (r - 1)$. For our example of $r = 4$, $d = 4$, we will have at most 12 drops with four balls on $n = 128$ floors! In fact, we will have at most 12 drops even for $n = 255$.

We need an interactive program, implementing the algorithm above, that will help us determine exactly the floors from which to drop balls for arbitrary n and d, so we can efficiently determine the hardness coefficient of the given balls. This program will take as input n and d, and tell us from what floor to drop the first ball. Then, depending on the result—break or no break—that we give it, the program will either tell us a new floor to drop a (new) ball from, or tell us what the hardness coefficient is. The program terminates only when the hardness coefficient has been determined, and it should tell us how many drops were needed total.

Here's the code for our program:

```
1.  def howHardIsTheCrystal(n, d):
2.      r = 1
3.      while (r**d <= n):
4.          r = r + 1
5.      print('Radix chosen is', r)
6.      numDrops = 0
7.      floorNoBreak = [0] * d
8.      for i in range(d):
9.          for j in range(r-1):
```

```
10.                     floorNoBreak[i] += 1
11.                     Floor = convertToDecimal(r, d, floorNoBreak)
12.                     if Floor > n:
13.                         floorNoBreak[i] -= 1
14.                         break
15.                     print ('Drop ball', i+1, 'from Floor', Floor)
16.                     yes = input('Did the ball break (yes/no)?:')
17.                     numDrops += 1
18.                     if yes == 'yes':
19.                         floorNoBreak[i] -= 1
20.                         break
21.             hardness = convertToDecimal(r, d, floorNoBreak)
22.             return hardness, numDrops
```

Lines 2–5 determine the radix `r` that we need to use. We will use a list of numbers, each of which will vary from `0` to `r-1` as our representation for the floor. Our `d`-digit list representation `floorNoBreak` has the most significant digit to the left, that is, index `0`, and is initialized to all `0`'s on line 7. Line 8 begins the outer **for** loop corresponding to the `d` phases, and line 9 begins the inner **for** loop corresponding to the drops of the chosen ball for the current phase.

We simply increment the appropriate digit in `floorNoBreak` corresponding to the phase we are in on line 10. Given that `r**d` can be significantly larger than `n`, we need to check that we don't generate drops from floors larger than `n`—this is done in lines 11–14. If the increment results in a floor higher than `n`, we are done with this phase and we immediately move to the next phase. That is what the **break** statement on line 14 does; loop iterations end immediately, and the line immediately after the loop is executed. In this case it is line 8, since we are breaking out of the inner **for** loop. Note that each **break** statement breaks out of the innermost loop it is enclosed in; iterations in outer loops will continue. Line 11 invokes a simple function—which we will present later—to convert from radix `r` to decimal. If we are moving to the next phase, we need `floorNoBreak` to correspond to the highest floor from which an actual drop has not resulted in a break. This is why we decrement `floorNoBreak` on line 13 before breaking out of the inner **for** loop.

We tell the user to drop a particular ball from a particular floor and wait for the result to be input by the user (lines 15–16). If the ball does not break, we merely continue the loop. If the ball breaks, we need to set `floorNoBreak` to be the highest floor from which a drop has not resulted in a break, which requires a decrement, as described above. We move to the next phase by breaking out of

the inner **for** loop (line 20).

Once we are done with all the phases, we have the hardness coefficient computed in floorNoBreak (line 21).

The function convertToDecimal shown below takes a base-r, d-digit list representation and returns the decimal equivalent.

```
1.  def convertToDecimal(r, d, rep):
2.      number = 0
3.      for i in range(d-1):
4.          number = (number + rep[i]) * r
5.      number += rep[d-1]
6.      return number
```

If we run the example we provided earlier:

```
howHardIsTheCrystal(128, 4)
```

we get the expected execution for the italicized yes/no user inputs below:

```
Radix chosen is 4
Drop ball 1 from Floor 64
Did the ball break (yes/no)?:no
Drop ball 1 from Floor 128
Did the ball break (yes/no)?:yes
Drop ball 2 from Floor 80
Did the ball break (yes/no)?:no
Drop ball 2 from Floor 96
Did the ball break (yes/no)?:no
Drop ball 2 from Floor 112
Did the ball break (yes/no)?:yes
Drop ball 3 from Floor 100
Did the ball break (yes/no)?:no
Drop ball 3 from Floor 104
Did the ball break (yes/no)?:no
Drop ball 3 from Floor 108
Did the ball break (yes/no)?:yes
Drop ball 4 from Floor 105
Did the ball break (yes/no)?:no
Drop ball 4 from Floor 106
Did the ball break (yes/no)?:no
Drop ball 4 from Floor 107
Did the ball break (yes/no)?:yes
```

The program returns a hardness coefficient of 106, and 11 drops.

Reducing the Number of Drops for Two Balls

Our algorithm drops balls from floors that are evenly distributed: k, $2k$, $3k$, and so on. Let's look at what was lost because of this assumption. For $n = 100$, with two balls and a hardness coefficient of 65, our algorithm drops the first ball from floors 11, 22, 33, 44, 55, and 66. If the first ball breaks when dropped from floor 66, the algorithm drops the second ball from floor 56, then 57, all the way to floor 65. When the ball does not break at floor 65, it reports the hardness coefficient as 65. This requires 16 drops in total. If the hardness coefficient was 98, then our algorithm would require 19 drops in total.

The example above shows us that our coded algorithm does not perfectly match the strategy we derived for $d = 2$. The square root of 100 is 10, so why did the algorithm choose $k = 11$? If we had chosen the radix r to be 10, then with 2 digits we can only represent floors up to 99, not 100. Therefore, the algorithm chooses the radix to be 11.[3]

Neither 10 nor 11 correspond to an optimal strategy—the optimal strategy requires uneven distribution of floors from which balls are dropped. Distributing the floors carefully and unevenly can shrink the number of drops required to a *maximum* of 14 for a 100-floor building. Let's say we want a maximum of k drops. For our *first* drop, if we drop the (first) ball from floor k, and it breaks, we require at most $k - 1$ drops of the second ball to find out from which floor the balls break. If the first ball does not break when dropped from floor k, we next drop this ball from floor $k + (k - 1)$. Why? If the ball breaks, we require $k - 2$ drops to discover the lowest floor from which the ball breaks. This would be a total of k drops, 2 for the first ball and $k - 2$ for the second ball.

Going on like this, we can relate k to the number of floors in the building, n.

$$n \leq k + (k - 1) + (k - 2) + (k - 3) + \ldots + 2 + 1$$

This means that $n \leq k(k + 1)/2$. For n = 100, $k = 14$. We should drop the balls from floors 14, 27, 39, 50, 60, 69, 77, 84, 90, 95, 99, and 100. For example, if the first ball breaks from floor 99 on the eleventh drop, we drop the second ball from floor 96, 97, and 98 in the worst case.

Exercises

Exercise 1: If you run howHardIsTheCrystal(128, 6), you see:

```
Radix chosen is 3
Drop ball 2 from Floor 81
```

The first drop uses ball 2. What happened is that $2^6 < 128$, so r = 3 is chosen. But

not only is $3^6 > 128$, so is $3^5 = 243$. Our algorithm skips over the first ball, since the first digit in our representation is always 0. Fix the code so it removes unnecessary balls and informs the user about the number of balls it is actually using. Your modified program should always start with dropping ball 1 from some floor.

Exercise 2: Modify the code so it prints the number of balls that were broken.

Exercise 3: Modify the code to print out the interval of floors currently under consideration. Initially, the interval is $[0, n]$. This interval keeps shrinking as ball drop results are entered into the program. Your code should print the new interval each time the user enters a result. The hardness coefficient corresponds to the final interval with a size of one floor.

Notes

1. While the Shanghai Tower is listed as having 127 usable floors, we'll assume that we can drop a ball off the roof, and call it a 128-floor tower!

2. We'll assume that terminal velocity does not come into play.

3. For related reasons, in the 128-floor case with 2 balls, the algorithm rounds up the square root of 128 to 12. However, this choice has a worst-case number of drops of 21 as well, which arises for a hardness coefficient of 119.

6

Find That Fake

When you see a fork in the road, take it.
—Yogi Berra

Programming constructs and algorithmic paradigms covered in this puzzle: Case analysis. Divide and conquer.

There are lots of popular puzzles about finding a counterfeit coin in a set of coins using only a weight balance. One variant is as follows: Find a fake coin in a set of nine identical-looking coins. You know that the fake coin is heavier than the rest. Your goal is to minimize the number of weighings.

The coins look identical and the fake is only slightly heavier than the rest. So if you put a differing number of coins on either side of the balance, the balance will never say that the two sides are equal. (Balance shown below.)

How many weighings do you need?

Given nine coins, of which eight are identical and one is slightly heavier, we can imagine choosing a coin arbitrarily and comparing it with each of the other eight coins. This guarantees that we will find the fake in eight weighings.

However, we can do much better through a process of aggregating coins and comparing collections of coins. Rather than eliminating one coin at a time as in the iterative solution above, we can eliminate many coins from consideration with one weighing. The strategy we will use falls into the category of divide-and-conquer approaches to puzzles and other problems.

Divide and Conquer

Suppose we pick four coins from the set of nine, and split them into two pairs of coins. If we weigh the pairs against each other, we have three possibilities:

. The pairs are equal in weight. This means that none of these four coins is a fake, and the fake is in the remaining five coins.
. The first pair is heavier. This means that one of the coins in this pair is a fake. We can determine which one is fake by comparing the coins in the first pair.
. The second pair is heavier. This situation is similar to case 2 above.

In cases 2 and 3 we can determine the fake coin using two weighings.

In case 1, after one weighing we are left with five coins, and one of these five is a fake. Let’s arbitrarily pick four coins out of the five and repeat the process. This second weighing of two versus two can result in three cases, just like the first. We will refer to these cases as case 1.1, case 1.2, and case 1.3. The 1 means the first weighing resulted in case 1, and the .1, .2, and .3 suffixes correspond to cases resulting from the second weighing.

In the second weighing, if the two pairs are equal in weight—case 1.1—the fake coin is indubitably the coin we did not pick, and we have found it in two

weighings.

Case 1.2 is the case where the first pair is heavier. After two weighings, we are not done yet. We still have to compare one coin against another, and this third weighing will tell us which one is fake. The same analysis holds for case 1.3. Three weighings to find the fake in the worst case is much better than eight, but is it optimal?

Suppose we pick two sets of three coins each from the initial set of nine coins. Let's compare their weights. As before, we have three cases:

. The sets are equal in weight. This means that none of these six coins is a fake, and the fake is in the remaining three coins.
. The first set is heavier. This means that one of the coins in this pair is a fake.
. The second set is heavier. This situation is similar to case 2 above.

This divide-and-conquer strategy is much more symmetric. In all three cases we have identified a set of three coins that has the fake. This has reduced the number of coins we have to deal with from nine to three using only one weighing. If we split the three coins into three sets of one coin each, and repeat this process, after the second weighing we will have determined the fake coin.

Let's say we have nine coins, numbered 0 through 8, and coin 4 is the fake. We will compare coins 0, 1, and 2 against 3, 4, and 5, and discover that 3, 4, and 5 are heavier. This is case 3 above. We will compare coin 3 versus 4 and discover that 4 is heavier, and therefore the fake.

Recursive Divide and Conquer

What happens if you have more than nine coins? Suppose we have twenty-seven coins and one of them is a fake, and slightly heavier. We split the coins into three groups of nine coins each. We follow the strategy above and find the group of nine that is heaviest. This is a group of nine with one fake coin, and we just solved that problem using two weighings. So we need only one more weighing to find a fake in twenty-seven coins, as compared to finding a fake in nine coins. As you can see, this divide-and-conquer strategy is very powerful, and we will keep returning to it in various puzzles, including the next one.

Now, let's write a program to solve the fake coin problem. In this particular instance of moving from the physical world to the virtual world of computer programs, we will have to pretend that we can't do something we really can. More on that later.

Given groups of coins, we can compare two groups to determine which one is heavier or whether they are equal. The function below is our weight balance.

```
1.  def compare(groupA, groupB):
2.      if sum(groupA) > sum(groupB):
3.          result = 'left'
4.      elif sum(groupB) > sum(groupA):
5.          result = 'right'
6.      elif sum(groupB) == sum(groupA):
7.          result = 'equal'
8.      return result
```

The function **sum** simply adds up the elements of the argument list and returns the sum. The function compare tells us whether the left side is heavier, the right side is heavier, or the sides are equal. Note that on line 6 we have an **elif** where an **else** would suffice. If the predicates for the **if** (line 2) and the first **elif** (line 4) are not true, then the groups will have equal weight. We used an **elif** on line 6 to explicitly indicate that this case corresponds to the two groups having equal weight.

Given a list of coins, we can split the list into three equal-sized groups. We will assume that the number of coins in the argument list for the following function is 3^n for some n.

```
1.  def splitCoins(coinsList):
2.      length = len(coinsList)
3.      group1 = coinsList[0:length//3]
4.      group2 = coinsList[length//3:length//3*2]
5.      group3 = coinsList[length//3*2:length]
6.      return group1, group2, group3
```

Lines 3–5 split the list into three different sublists using integer division and the slice operation in Python. The operator // is integer division, for example, 1//3 = 0, 7//3 = 2, 9//3 = 3, and 11//3 = 3. Recall that if we have b = a[0:3], the first three elements of a are copied into list b, which will have length 3. (We will assume that list a has length at least 3.) In lines 3–5, the first third of the list coinsList is copied to group1, the second third to group2, and the last third to group3. Line 4 is the most complex: If length is 9, then length//3 = 3, length//3*2 = 3*2 = 6, since // takes precedence over *.

All these sublists are returned (line 6).

The following function does one comparison, that is, one weighing (line 2), and determines which group has the fake coin based on the result. The function

assumes that the fake coin is heavier than the other coins, which are all identical.

```
1.    def findFakeGroup(group1, group2, group3):
2.        result1and2 = compare(group1, group2)
3.        if result1and2 == 'left':
4.            fakeGroup = group1
5.        elif result1and2 == 'right':
6.            fakeGroup = group2
7.        elif result1and2 == 'equal':
8.            fakeGroup = group3
9.        return fakeGroup
```

If the left side is heavier (line 3), then the first group contains the fake coin (case 2 of the algorithm). If the right side is heavier (line 5), then the second group contains the fake coin (case 3). If the two sides are equal (line 7), then the fake coin is contained in the third group (case 1). Note that we again used an **elif** where an **else** would suffice, on line 7.

Now we are ready to code our divide-and-conquer algorithm, which finds the fake coin in a list of 3^n coins whose weights are contained in the argument coinsList.

```
1.    def CoinComparison(coinsList):
2.        counter = 0
3.        currList = coinsList
4.        while len(currList) > 1:
5.            group1, group2, group3 = splitCoins(currList)
6.            currList = findFakeGroup(group1, group2, group3)
7.            counter += 1
8.        fake = currList[0]
9.        print ('The fake coin is coin',
                  coinsList.index(fake) + 1,
                  'in the original list')
10.       print ('Number of weighings:', counter)
```

Line 2 initializes the variable counter, which counts the number of weighings that will be performed. Line 3 simply creates a new reference to coinsList that will be used to point to the current collection of coins under consideration. Lines 4–7 correspond to the divide-and-conquer strategy that shrinks coinsList to a third of its size at each iteration. When the size of currList is 1, we have found the fake. The algorithm is as described earlier—split coins into three groups in splitCoins, then determine the group with the fake coin by comparing group1 to group2.

At line 8 we have exited the **while** loop, implying **len**(currList) is 1 (assuming it

was 3^n for some *n* to begin with). We then print out the position of the fake coin in coinsList and the number of weighings given by the value of counter.

Let's say we have the set of coins represented by coinsList below:

```
coinsList = [10, 10, 10, 10, 10, 10, 11, 10, 10,
        10, 10, 10, 10, 10, 10, 10, 10, 10,
        10, 10, 10, 10, 10, 10, 10, 10, 10]
```

If we run:

```
CoinComparison(coinsList)
```

We get:

```
The fake coin is coin 7 in the original list
Number of weighings: 3
```

How is this fake found? In the first weighing in the call to findFakeGroup we compare the first row of nine coins in coinsList, that is, group1, with the second row, group2. The balance will return 'left' and therefore the group chosen is group1. In the next weighing, the first three coins of the first row of coinsList are compared against the next three. Since these are equal, findFakeGroup will return group3, that is, the last three coins. The last weighing compares the fake coin against an authentic one and returns 'left', and thus we find the fake.

We can just look at each of the elements of coinsList and determine which coin is heaviest by scanning the list and comparing numeric values. However, this will require—in the worst case—looking at each coin and comparing values to find the one value different from all others. Our divide-and-conquer algorithm is designed to minimize the number of comparisons because we assume in this puzzle that comparisons using the weight balance are expensive. For example, coinsList may be a data structure that costs a lot to access since it is stored in a faraway computer. All the faraway computer can do is compare sets of coins that you specify and tell you which set is heavier. Transmission of information between your local computer and the faraway computer is expensive, and needs to be kept to a minimum.

What if you didn't know whether the coin was heavier or lighter in findFakeGroup? We need to write a findFakeGroupAndType function. Below is what you have to do for the case of result1and2 == 'left', that is, the case where group1 is heavier than group2 in findFakeGroupAndType. The other cases will need similar modifications.

```
1.      if result1and2 == 'left':
```

```
2.      result1and3 = compare(group1, group3)
3.      if result1and3 == 'left':
4.          fakeGroup = group1
5.          type = 'heavier'
6.      elif result1and3 == 'equal':
7.          fakeGroup = group2
8.          type = 'lighter'
```

When group1 is heavier than group2, we compare group1 and group3 (line 2). If group1 is heavier than group3, it is clear that group1 contains the fake and the fake is heavier than the genuine coin. If group1 is equal in weight to group2 (line 6), then group2 contains the fake and the fake is lighter than the genuine coin. Note that if there is exactly one fake (that is lighter or heavier) we can't have result1and3 == 'right', because this means that group1 is heavier than group2, and that group3 is heavier than group1 and group2 by transitivity, which is a contradiction, since we are guaranteed with exactly one fake that two groups will have equal total weight.

Fortunately you only have to do this extra work once, right at the beginning. In two weighings you find a group that is one-third the size of the coin collection you are given, and you know whether the fake in the group is heavier or lighter. Then you can proceed as before. Of course, if we know the coin is lighter, we need to return a different group than what is returned by findFakeGroup, which assumes the fake coin is heavier.

Therefore, if you have 3^n coins and one of them is fake, and is lighter or heavier, but you don't know which, you only need $n + 1$ weighings to determine the fake coin. Fleshing out the described strategy is one of the exercises of this puzzle chapter.

Ternary Representation

The non-decimal number representation was very helpful in the crystal ball puzzle (puzzle 5), and it is natural to ask whether such a representation can be used to solve our fake coin problem. Assume we have 3^n coins. We'll use a ternary representation, and the coins will be numbered from 0 to $3^n - 1$ with the numbers represented in ternary. If $n = 4$, the first coin is represented as 0000 and the last one as 2222. For the first weighing, we will simply select three groups of coins corresponding to the first digit in the representation being a 0, 1, and 2. Each group in this first weighing will have nine coins. Let's assume we determine that the fake coin is in the 2 pile. For the second weighing, we will

select the three groups corresponding to the first two digits being 20, 21, and 22. Four weighings will specify all four digits, for example producing 2122, meaning that we are down to one coin, the fake one.

A Popular Variant Weighing Puzzle

Suppose we have twelve coins that look identical, but one is fake and is slightly heavier or lighter than the others—you don't know which. Can you find out which one is fake using at most three weighings? This problem is obviously more difficult than the nine-coins puzzle since there are more coins. Our algorithm takes three weighings for nine coins. You will have to compare differing numbers of coins and repeatedly weigh some coins.

Exercises

Exercise 1: The code assumes there is one coin that is heavier than the rest. Given a coins list where all coins have the same weight, it claims that coin 1 is the fake. Fix the code so it does the right thing and says that none of the coins are fake.

Puzzle Exercise 2: Write CoinComparisonGeneral, which can determine the fake without assuming that the fake is heavier, as the current code does. This will involve filling in the other two cases for result1and2 in findFakeGroupAndType and then calling findFakeGroupAndType for the first split, and either findFakeGroupHeavier or findFakeGroupLighter for the remaining splits. The given procedure findFakeGroup assumes that the fake coin is heavier, and you will need to write a companion procedure that assumes the fake coin is lighter and returns the appropriate group. Your code should handle the case of there being no fake coin.

Puzzle Exercise 3: Suppose there are two fake coins, which are identical and heavier than the rest. Modify CoinComparison to find *one* of the fake coins—it doesn't matter which one. Note that if you have two groups of coins, with each group having a fake, the balance is going to say that the groups are equal in weight. If you pick the third group (that was not in the comparison) you will miss finding the fake. Don't bother with correctly counting the number of weighings in your modified code.

7

Hip to Be a Square Root

If the tree's root is in the shape of an "i," then would that make this tree an imaginary lumber?
—Author unknown

> Programming constructs and algorithmic paradigms covered in this puzzle: Floating-point numbers and arithmetic. Continuous domain bisection search. Discrete binary search.

You are asked to find the square roots of a set of numbers. Some students learn the long division–like approach to finding square roots in middle school—our approach is going to have nothing in common with the long division approach.

Iterative Search

If you know the number *n* is a perfect square, you can try starting with 1 and squaring that, and if the square is less than *n*, increment to 2, and so on. When you get to the square root *a*, you see that $a^2 = n$ and you stop. This guessing and checking works reasonably well, especially with fast modern computers, but has its limitations, as we will see. But first, here's code that does the guessing and checking:

```
1.  def findSquareRoot(x):
2.      if x < 0:
3.          print ('Sorry, no imaginary numbers!')
```

```
4.              return
5.          ans = 0
6.          while ans**2 < x:
7.              ans = ans + 1
8.          if ans**2 != x:
9.              print (x, 'is not a perfect square')
10.             print ('Square root of ' + str(x) +
                       ' is close to ' + str(ans - 1))
11.         else:
12.             print ('Square root of ' + str(x) +
                       ' is ' + str(ans))
```

Line 1 defines the function with a single argument `x`. Lines 2–4 check that `x >= 0` before proceeding. The important code is the **while** loop in Lines 5–7. It initializes the guess `ans` to `0` and then squares `ans`—as you might have guessed, `**` is the raise-to-power operator—to check if the square is less than `x`. If so, we enter the body of the **while** loop (line 7), which increments `ans` (i.e., adds `1` to `ans`). When `ans**2 >= x`, we exit the **while** loop and move to line 8.

If the square of `ans` is exactly `x`, we have found the square root of the perfect square `x`, which has an integer square root. If not, we have found an `ans` such that `(ans - 1)**2 < x < ans**2`. Why is this the case? When we execute the body of the **while** loop—namely, line 7—for the last time, the square of the value of `ans` prior to executing the increment operation is less than `x`, otherwise we would not have entered the loop body. Once this increment occurs, the loop condition is no longer **True**, so we exit the loop and execute line 8.

If we run `findSquareRoot(65536)` we get:

```
Square root of 65536 is 256
```

If we run `findSquareRoot(65535)` we get:

```
Square root of 65535 is close to 255
```

Suppose we want to get closer than `255` to the actual square root of `65535`. The square root—allowing for fractional roots—is `255.998046868`, which is a lot closer to `256` than `255`. You might think we could have easily reported `256` by replacing `str(ans - 1)` with `str(ans)` on line 10. But for `findSquareRoot(65026)` we are better off reporting `ans - 1 = 255` since `255` is much closer to the square root of `65026` than `256`.

We need to change the **while** loop in lines 5–8 to increment by something much smaller than `1`. The code below allows a user to determine how close the square root should be and how much to increment each guess by.

```
1.  def findSquareRootWithinError(x, epsilon, increment):
2.      if x < 0:
3.          print ('Sorry, no imaginary numbers!')
4.          return
5.      numGuesses = 0
6.      ans = 0.0
7.      while x - ans**2 > epsilon:
8.          ans += increment
9.          numGuesses += 1
10.     print ('numGuesses =', numGuesses)
11.     if abs(x - ans**2) > epsilon:
12.         print ('Failed on square root of', x)
13.     else:
14.         print (ans, 'is close to square root of', x)
```

There are a few changes from the first version. We can stop when we have an answer that is within epsilon and so the **while** loop on line 7 has a different entry condition. We are incrementing by increment, specified in the function invocation as an argument. We also added a mechanism to count the number of guesses, that is, the iterations in the **while** loop. This will come in handy during performance analysis.

A final point is the **abs** on line 11, which finds the absolute value of its argument. We do not want to find a square root ans whose square is *bigger* or smaller than x by more than epsilon. How could the square be bigger? The last time the **while** loop is entered, ans**2 is clearly smaller than x. However, once ans has been raised in value by increment, it is entirely possible that ans**2 is greater than x, and further than epsilon away from x. This happens because we have chosen too large a value for increment, as we will illustrate.

If the code above is run with different arguments, here are the results:

```
>>> findSquareRootWithinError(65535, .01, .001)
numGuesses = 255999
Failed on square root of 65535
>>> findSquareRootWithinError(65535, .01, .0001)
numGuesses = 2559981
Failed on square root of 65535
>>> findSquareRootWithinError(65535, .01, .00001)
numGuesses = 25599803
255.99803007 is close to square root of 65535
```

The error epsilon is set to .01 in all the runs. The first few runs fail because the

while loop “jumps” over the answer. In the last iteration of the loop, when the **while** condition is **True** for the last time, x - ans_0**2 is greater than epsilon, where ans_0 represents the value of the variable ans when the condition is tested and found to be **True**. Inside the loop, the variable ans is incremented by increment. At this point,

x - (ans_0 + increment)**2 < - epsilon

or

(ans_0 + increment)**2 - x > epsilon

and we fail. If we choose a small enough value for increment, then we step through various answers with fine enough resolution that we find a square root that is within epsilon. This has the unfortunate downside of increasing the runtime of the program—the number of guesses required is a proxy for the runtime, and is over 25 million for a successful search.

Assume that x >= 1. Let’s say the initial search space for the square root is [0, x]. So far, we have split the search space up into “slots,” each of size increment. So the number of slots we have to search is x/increment. We guess that the square root is in one particular slot, and if not, we go to the next, and so on. As increment shrinks, our runtime increases.

A word of caution: Be careful when checking the equality of floating-point numbers. Numbers you think are equal may not be! For example, 0.1 + 0.2 produces 0.30000000000000004 on most machines using IDLE and Python 3.5. This produces horrifying behaviors like:

```
>>> x = 0.1 + 0.2
>>> x == 0.3
False
```

In our code we did not check for strict equality of floating-point numbers, so we avoided the above behavior. This is a good rule to follow in general.

We can improve this runtime dramatically using notions similar to how we searched for the counterfeit coin. The same algorithmic divide-and-conquer technique that allowed us to find the counterfeit coin among 3^n coins with n weighings can be applied here, albeit in a different, perhaps unrecognizable, form.

Can you think of a way to improve the runtime?

Bisection Search

Here's the key insight: If we have a particular guess, call it a, and we realize that a**2 > x, then we never need to look at any guesses greater than a. Similarly, if we have a particular guess a, and a**2 < x, then we never have to look at any guesses less than a. Rather than searching sequentially, slot by slot, why not shrink the range [0, x]? In the two cases just described, we can shrink the range to [0, a] and [a, x], respectively.

If this notion of shrinking the search space based on an experiment seems familiar, it might be because you saw it in puzzle 5, the crystal ball puzzle, and puzzle 6, the fake coin puzzle.

What value of a should we guess? It makes sense to guess a value that is midway between 0 and x. And similarly, when we have a range [0, a] or [a, x], it makes sense to guess a new value that is midway between the two extremes. This is what the bisection search code (below) does.

```
1.  def bisectionSearchForSquareRoot(x, epsilon):
2.      if x < 0:
3.          print ('Sorry, imaginary numbers are out of scope!')
4.          return
5.      numGuesses = 0
6.      low = 0.0
7.      high = x
8.      ans = (high + low)/2.0
9.      while abs(ans**2 - x) >= epsilon:
10.         if ans**2 < x:
11.             low = ans
12.         else:
13.             high = ans
14.         ans = (high + low)/2.0
15.         numGuesses += 1
16.     print ('numGuesses =', numGuesses)
17.     print (ans, 'is close to square root of', x)
```

Lines 9–14 correspond to bisection search. The initial interval is [low, high], and we guess ans as the midpoint of the interval. Depending on where the square root lies, we move to [low, ans] or [ans, high]. Then we use the midpoint of the new interval as our next guess.

When the above code is run with x = 65535 and epsilon = 0.01:

```
bisectionSearchForSquareRoot(65535, .01)
```

we get:

```
numGuesses = 24
255.998046845 is close to square root of 65535
```

Using 24 guesses (iterations of the **while** loop) we get an answer that is the same, to four decimal places, as what we got with over 25 million guesses using sequential search. Both answers are correct in that the squares of the answers are within epsilon = .01 of 65535. This is a spectacular improvement, reminiscent of being able to find a counterfeit coin out of 3^n coins using n weighings. We are splitting the search space in half with each guess, so after 24 guesses the search space is $1/2^{24}$ the size of the original [0, 65535] space. And $65535/2^{24}$ is less than .01, so we converge after 24 splits!

How do the intervals shrink as we make guesses? Initially, the interval is [0.0, 65535.0], and the guess is (0.0 + 65535.0)/2 = 32767.5. This is obviously too large, and the interval shrinks to [0.0, 32767.5]. It keeps shrinking until we get to the interval [0.0, 511.9921875], with a guess of 255.99609375. This guess is too small and the interval becomes [255.99609375, 511.9921875]. The guess for this interval is 383.994140625, which is too large. After a few more iterations, the guess goes down to 255.99804684519768, which is within the error bound, and the loop terminates.

Binary Search

Bisection search works because of a monotonicity property of the square and positive square root functions. That is, if a number $x > y \geq 0$, then $x^2 > y^2$. Thus, if our guess a is such that a**2 > x, then we never need to look at any guesses greater than a. In the crystal ball puzzle, puzzle 5, we relied on the very same property when we assumed that if a ball breaks when dropped from floor *f*, it (or a different identical ball) will also break when dropped from any higher floor. And similarly, if a ball does not break when dropped from floor *f*, it will not break when dropped from any lower floor.

Bisection search over continuous variables is intimately related to the concept of binary search over discrete variables, which we will look at now.

We have a list of numbers and we want to know if a particular number is in the list or not. When we write

```
member in myList
```

we are checking if member is contained in myList. How is this done? One way is to look at each entry in myList in sequence and check if the entry is equal to member, as shown below.

```
1.  NOTFOUND = -1
2.  Ls = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
          53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
3.  def lsearch(L, value):
4.      for i in range(len(L)):
5.          if L[i] == value:
6.              return i
7.      return NOTFOUND
```

lsearch(Ls, 13) will return 5, since Ls[5] = 17. Remember that Python list indices start with 0.

If we call lsearch(Ls, 26) we will get NOTFOUND. In the worst case—the number not being in the list L—we will look at **len**(L) entries. Incrementing i in the **for** loop above is very similar to incrementing by increment in the iterative search of findSquareRootWithinError.

We would like to do better—that is, look at far fewer entries—to either find the number or determine that the number is not in the list. If the list is sorted, as in our example of list L, we can do much better.

Sorting gives us a monotonicity property (sound familiar?) that we can exploit by using binary search. While we are running on Ls in the example below, we are referring to the argument name L as we describe what the algorithm does.

. When we look for 26, we first look at the midpoint of the list, L[12], which is 41. Since 41 > 26, we know that 26 cannot be in the part of the list after 41. If it is in the list L, it will be at an index of 11 or less.
. We next look at L[5], which is 13. Since 13 < 26, we now need to look in between L[7] and L[11].
. We look at L[8], which is 23. Since 23 < 26, we need to look between L[9] and L[11].
. We look at L[10], which is 31. Since 31 > 26, we need to look at indices less than 10.
. We only have L[9] to look at, and that is 29. This means that what we are searching for is not in the list, and we return NOTFOUND.

Rather than looking at 25 L[i] entries to determine that the number 26 is not in L, we have only looked at 5 different L[i] entries. For a general list L, we will only require $\log_2$ **len**(L) checks in the worst case, to either find the number or determine that it does not exist. **len**(L) = 25 should be increased to the next power of 2, namely 32, and $\log_2 32 = 5$.

Let's look at an example where the number is in the list L, as with the number 29 in list Ls.

. We look at the midpoint L[12], which is 41. And 41 > 29.
. We look at L[5], which is 17. And 17 < 29.
. We look at L[8], which is 23. And 23 < 29.
. We look at L[10], which is 31. And 31 > 29.
. We look at L[9], which is 29, and return the index 9. (Note that if L[9] happened to be 28 instead of 29, we would return NOTFOUND since we have narrowed the search down to L[9].)

Here's code for binary search on a sorted list.

```
1.  def bsearch(L, value):
2.      lo, hi = 0, len(L) - 1
3.      while lo <= hi:
4.          mid = (lo + hi) // 2
5.          if L[mid] < value:
6.              lo = mid + 1
7.          elif value < L[mid]:
8.              hi = mid - 1
9.          else:
10.             return mid
11.     return NOTFOUND
```

As you can see, this is very similar to the bisection search code. The interval of search is initially the entire list (line 2) and is represented as lo, hi, which are indices. The **while** loop begins on line 3, and as long as the interval has at least one entry in it, the search continues. If the value at L[mid] is the value we are searching for, we can return it. When lo equals hi, there is only one entry in the interval and we have one last iteration of the loop—either we find the value, or lo becomes greater than hi.

Finally, note that mid = (lo + hi)//2 is integer division, which truncates fractional values. So if lo = 7 and hi = 8, we will get mid = 7. The two examples prior to the

bsearch code match its execution.

Ternary Search?

Binary search means splitting the search interval in two through one comparison. Given an interval [0, n-1], we look at the value of L[n//2] and decide if we want to continue with [0, n//2-1] or [n//2+1, n-1]. We could check two positions, for example, L[n//3] and L[2n//3], and choose an interval whose size is 1/3 of the original depending on the two comparisons. The number of comparisons required in the worst case for binary search is $\log_2 n$, and for ternary search it's $2 \log_3 n$, which is larger. Ternary search is useful in our fake coin puzzle (puzzle 6) because we only had to do one weighing (comparison) to reduce the number of coins to one-third of the original.

Exercises

Exercise 1: The bisection search program fails for x = 0.25 or for any x < 1 - epsilon. That is, it doesn't finish. Can you diagnose and fix it?

Hint: Think of what the square root of 0.25 is, and what range the program is searching.

Exercise 2: Modify the procedure bsearch to take an additional argument: the length of the interval in which an iterative sequential search is conducted. The current procedure continues with binary search until the element is found. However, it may be faster to not use division, and use sequential search within the interval [lo, hi] when the length of the interval hi - lo is smaller than a specified length.

Exercise 3: Modify the bisection search program so it finds a root of the function $x^3 + x^2 - 11$ to within a given error (e.g., 0.01). You will need to start with a zero-crossing interval such as [−10, 10].

8

Guess Who Isn’t Coming to Dinner[1]

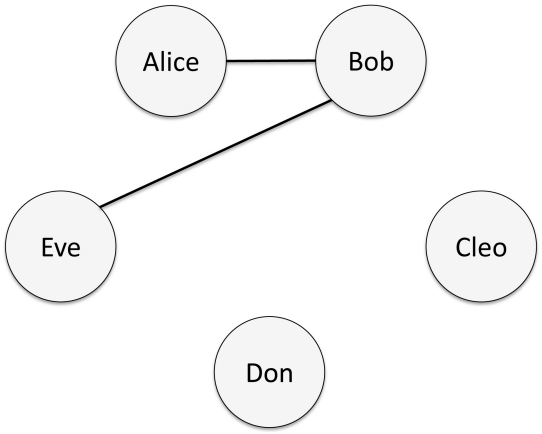
My friend and I had a two-hour fight on whether or not we were fighting.
—Author unknown

> Programming constructs and algorithmic paradigms covered in this puzzle: List concatenation. Exhaustive enumeration and encoding combinations.

You have a wide social circle and you like entertaining. Unfortunately, not all your friends like each other—in fact, some dislike others intensely. When you throw a dinner party, you want to make sure that no fights are going to break out. So you don’t want to invite any pair of friends who are liable to get into fisticuffs or heated arguments and spoil the party. But the more the merrier, so you want to invite as many friends as possible.

To get a sense of what you are up against, let’s represent your social circle as a graph. Each vertex of the graph is one of your friends. The graph also has edges between vertices. If Alice dislikes Bob, we will have an edge between the Alice vertex and the Bob vertex. Note that Bob may (secretly) like Alice or the dislike may be mutual, but either way, you don’t want Alice and Bob together at your house. The edge between the Alice and Bob vertices signifies that one of them does not like the other or that both dislike each other.

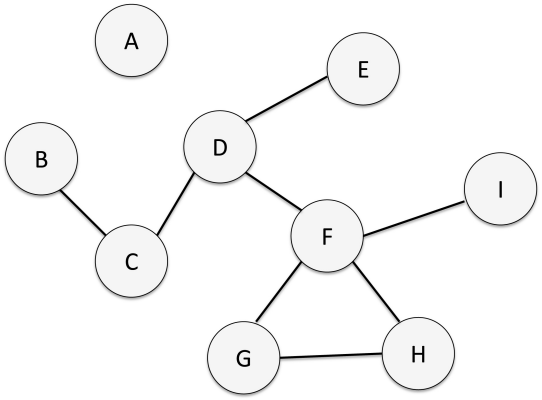
Let’s pretend that your current social circle looks like this:



The total number of possible guests is five. However, there are two edges in the above graph, representing “dislike” relationships. You can’t invite Alice and Bob together, and you can’t invite Bob and Eve. There is no transitivity in the dislikes relationship—Alice dislikes Bob, who dislikes Eve, but Alice is just fine with Eve, and Eve with Alice, in the example above. If Eve disliked Alice, or vice versa, there would be an edge between Eve and Alice. Likes or dislikes are unpredictable, just as in real-life social circles!

Cleo and Don are obvious choices since they don’t preclude anyone else from being invited. If you invite Bob, you can’t invite either Alice or Eve. But if you don’t invite Bob, you can invite four of your friends: Cleo, Don, Alice, and Eve. That is the maximum number of people you can have over for dinner in your current social circle.

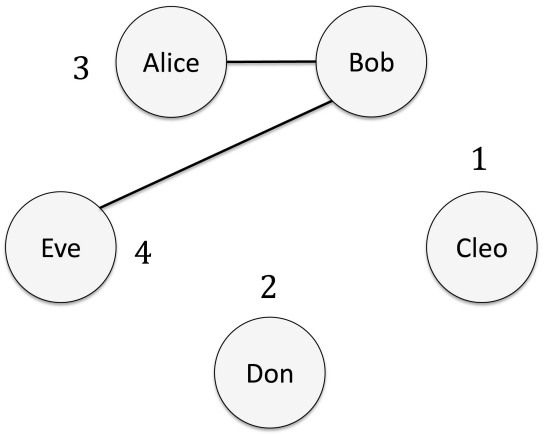
Given an arbitrarily complex social circle (e.g., the graph below), can you think of an algorithm that always finds a maximum number of mutually friendly people to invite for dinner?



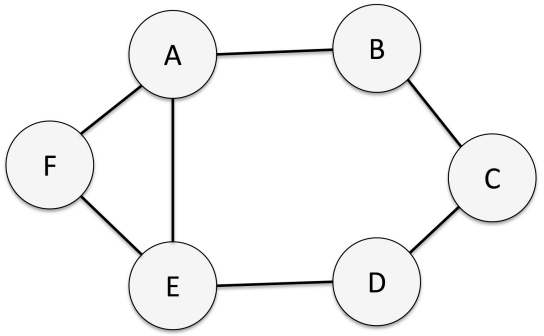
First Attempt

We'll first try what is called a greedy approach. A greedy algorithm for our puzzle would pick the guest with the fewest dislikes, that is, the one with the fewest connecting edges. Once this guest is picked, all guests disliked by this guest are eliminated from consideration. The process continues until all guests are either selected or eliminated.

For our first example, the greedy algorithm would pick Cleo and Don (no connecting edges). No guests are eliminated. Then, it would pick Alice or Eve, who each have one connecting edge, since Bob has two connecting edges. Picking either Alice or Eve eliminates Bob. The guest who was not picked remains, and can be picked. A possible sequence of greedy selections resulting in the maximum selection is shown below.



Now consider the problem below. There are several vertices/guests with two edges, and the greedy algorithm could easily pick *C*.



If the greedy algorithm picks *C*, then *B* and *D* are eliminated. After that it can only pick one of *A*, *E*, or *F*, producing a solution with two guests. However, the maximum selection has three guests, namely, *B*, *D*, and *F*.

A greedy approach is not guaranteed to work for our puzzle. In this book we

will look at other problems where greedy algorithms do and do not work.

Always Finding the Maximum Selection

Here's an approach that is exhaustive and guarantees optimality:

. Generate all possible combinations of guests to go to the dinner party, ignoring any dislike relationships.
. Check for combinations containing the pairs of people who dislike each other and remove those combinations.
. Out of the remaining viable combinations, find one with the maximum people. This is the optimum solution. There might be multiple combinations with the maximum number.

Let's go through these steps for your original social circle of five friends.

. Given *n* guests, we can choose to invite each guest, or not. There are 2 choices for each guest, and therefore 2^n different combinations. One of these combinations corresponds to inviting all the guests, and one corresponds to inviting none. For your social circle, $n = 5$, so we have 32 different combinations of guests, listed below. We have abbreviated the names to their initials and we refer to guests 'Alice,' 'Bob,' and so on, as A, B, for brevity.

[]
[A] [B] [C] [D] [E]
[A, B] [A, C] [A, D] [A, E] [B, C] [B, D] [B, E] [C, D] [C, E] [D, E]
[A, B, C] [A, B, D] [A, B, E] [A, C, D] [A, C, E] [A, D, E] [B, C, D] [B, C, E] [B, D, E] [C, D, E]
[A, B, C, D] [A, B, C, E] [A, B, D, E] [A, C, D, E] [B, C, D, E]
[A, B, C, D, E]

. We have two pairs of dislikes: [A, B] and [B, E]. We cannot have any combination that has both A and B in it, and we cannot have any combination that has both B and E in it. This leaves us with:

[]
[A] [B] [C] [D] [E]
~~[A, B]~~ [A, C] [A, D] [A, E] [B, C] [B, D] ~~[B, E]~~ [C, D] [C, E] [D, E]
~~[A, B, C]~~ ~~[A, B, D]~~ ~~[A, B, E]~~ [A, C, D] [A, C, E] [A, D, E] [B, C, D] ~~[B, C, E]~~ ~~[B, D, E]~~ [C, D, E]

~~[A, B, C, D]~~ ~~[A, B, C, E]~~ ~~[A, B, D, E]~~ [A, C, D, E] ~~[B, C, D, E]~~ ~~[A, B, C, D, E]~~

. Given all the viable combinations, there are none with five people, and there is exactly one with four people: [A, C, D, E]. Therefore, we pick this one, which corresponds to inviting Alice, Cleo, Don, and Eve.

This approach will produce the maximum cardinality selection [B, D, F] for our second example, where the greedy approach failed.

Our challenge now is to code this algorithm. We have to generate all combinations systematically, purge the unfriendly ones, and then pick a combination with the most guests. We'll look at code for each of these steps in turn.

Generating All Combinations

There are many ways, of course, to find all combinations (or subsets) of a set of strings, vertices, or people. We observe that each combination can be viewed as a number between 0 and $2^n - 1$. First, view the choice of whether or not the first object is in the combination as a binary digit (or bit), which can be 0 or 1. View the choice similarly for each of the other objects. Therefore, a string of all 0's is the empty combination, since none of the objects are in the combination. If all the objects are in the combination, we represent the combination as a string with all 1's. Given the ordering A, B, C, D, E, here are the strings for a few combinations, as well as the decimal values corresponding to the strings (when the strings are viewed as binary numbers):

[]	00000	0
[A, B]	11000	24
[B, C, D]	01110	14
[A, B, C, D, E]	11111	31

To generate all the combinations, we can go right to left in the table above and simply iterate through the numbers from 0 to $2^n - 1$.

```
1.  def Combinations(n, guestList):
2.      allCombL = []
3.      for i in range(2**n):
4.          num = i
5.          cList = []
```

```
6.                 for j in range(n):
7.                     if num % 2 == 1:
8.                         cList = [guestList[n - 1 - j]] + cList
9.                     num = num//2
10.                allCombL.append(cList)
11.            return allCombL
```

The function takes the number of guests, as well as the guest list, as input arguments. `n = 5` and `guestList = [A, B, C, D, E]` in our example. We want to iterate through the loop 2^n times, beginning with 0 to $2^n - 1$, incrementing by 1 each time.

Given a value of `i`, we generate an `n`-bit binary string corresponding to this value, copied into `num`. We do this copy because we will modify `num` and it is a no-no to modify a **for** loop counter inside the loop. The `n`-bit string generation is done by the inner **for** loop, which begins at line 6. Here are the steps to generate the binary string associated with `24`—we are using the remainder method:

```
24 % 2 = 0      24//2 = 12
12 % 2 = 0      12//2 = 6
 6 % 2 = 0      6//2 = 3
 3 % 2 = 1      3//2 = 1
 1 % 2 = 1      1//2 = 0
```

The bits are generated in order of increasing significance, meaning that we need to read the binary number bottom-up as `11000`. This explains why on line 8 we index `guestList` by `n - 1 - j`, since we want the most significant bit to correspond to the first guest in `guestList` (`A` in our example). Further, rather than appending to `cList`, we want to order the guests in the same order as in `guestList`. That is, we want the representation to be `['A', 'B']` and not `['B', 'A']`.[2] This is why we concatenate a list containing the guest—note the encircling `[]` around `guestList[n - 1 - j]`—with the prior list. If we did not have the encircling `[]` we would be trying to concatenate a string with a list, which would result in an error when we run the program. Concatenation in this order means that the `guestList` entries with lower indices will appear first in `cList`.

Note that `cList` is reset at the beginning of each outer **for** loop iteration on line 5, and once it is filled in, it is appended to `allCombL` on line 10.

Removing Unfriendly Combinations

The second step in our algorithm is to look at each combination and check that

the combination does not contain any of the dislike pairs. If both guests in the pair are in the combination, then the combination is unfriendly and should be discarded. The code below accomplishes this.

```
1.      def removeBadCombinations(allCombL, dislikePairs):
2.          allGoodCombinations = []
3.          for i in allCombL:
4.              good = True
5.              for j in dislikePairs:
6.                  if j[0] in i and j[1] in i:
7.                      good = False
8.              if good:
9.                  allGoodCombinations.append(i)
10.         return allGoodCombinations
```

Each combination is iterated over in the outer **for** loop, beginning with line 3. Then, each dislike pair is iterated over. Each dislike pair is a list with two guests, and in our example, we have two dislike pairs: ['A', 'B'] and ['B', 'E']. A combination with both 'A' and 'B' in it is unfriendly and needs to be discarded. Line 6 checks for this. We also have to check if the combination has 'B' and 'E'. Note that if we instead check j **in** i, this will always be **False**. Checking ['A', 'B'] **in** ['A', 'B', 'C', 'D', 'E'] returns **False** because j **in** i checks if one of the members of the list i is j. The list j = ['A', 'B'] is *not* a member of ['A', 'B', 'C', 'D', 'E'], even though each element of list j is in list i. (For completeness, note that ['A', 'B'] is a member and is **in** [['A', 'B'], 'C', 'D', 'E'].)

Choosing the Minimum Combination

Last, we have to find a combination with the maximum number of guests to invite for dinner. Below, we invoke the first two procedures in line 2 and lines 3 and 3a before finding a combination to invite.

```
1.      def InviteDinner(guestList, dislikePairs):
2.          allCombL = Combinations(len(guestList), guestList)
3.          allGoodCombinations = \
3a.                 removeBadCombinations(allCombL, dislikePairs)
4.          invite = []
5.          for i in allGoodCombinations:
6.              if len(i) > len(invite):
7.                  invite = i
8.          print ('Optimum Solution:', invite)
```

Lines 4–7 iterate through the good combinations to find one with the maximum number of guests. The first combination with the maximum number is printed. Python has many built-in functions for manipulating and processing lists, and lines 4–7 can be replaced by:

```
4a.         invite = max(allGoodCombinations, key=len)
```

max is a Python built-in function that returns the maximum element of the argument list. You can also specify a key, which is the function used to compare two elements. The key in our case needs to be the **len** function.

If we run the code:

```
dislikePairs = [['Alice','Bob'], ['Bob','Eve']]
guestList = ['Alice', 'Bob', 'Cleo', 'Don', 'Eve']
InviteDinner(guestList, dislikePairs)
```

The code prints out:

```
Optimum Solution: ['Alice', 'Cleo', 'Don', 'Eve']
```

But you already knew that. The motivation here was to try to solve more complex problems, like the graph with nine vertices we saw earlier. If we run the code:

```
LargeDislikes = [['B', 'C'], ['C', 'D'], ['D', 'E'], ['F', 'G'],
                 ['F', 'H'], ['F', 'I'], ['G', 'H']]
LargeGuestList = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
InviteDinner(LargeGuestList, LargeDislikes)
```

We get:

```
Optimum Solution: ['A', 'C', 'E', 'H', 'I']
```

Optimizing Memory Usage

The code we showed you creates and stores a list of length 2^n, where n is the number of guests. For large n, this can be quite a bit of memory. For our problem, we cannot avoid having to search a worst-case exponential number of combinations of guests, but we can easily avoid having to *store* an exponential-sized list.

The way we avoid storing the list is simple: We generate each combination exactly as before. Rather than storing the combination in a list, we immediately process it in the same way as before to determine whether it is a good or bad combination. If it is good, we compare it with the best, that is, maximum-length

combination we have seen thus far, and update the best combination as necessary.

Here's the code for the optimized version. You will recognize every line, since you have seen it before!

```
1.  def InviteDinnerOptimized(guestList, dislikePairs):
2.      n, invite = len(guestList), []
3.      for i in range(2**n):
4.          Combination = []
5.          num = i
6.          for j in range(n):
7.              if (num % 2 == 1):
8.                  Combination = [guestList[n-1-j]] + Combination
9.              num = num // 2
10.         good = True
11.         for j in dislikePairs:
12.             if j[0] in Combination and j[1] in Combination:
13.                 good = False
14.         if good:
15.             if len(Combination) > len(invite):
16.                 invite = Combination
17.     print ('Optimum Solution:', invite)
```

Lines 3–9[3] generate the combination corresponding to value `num = i`. Lines 10–13 determine whether the combination is a good one or not. Finally, lines 14–16 update the best combination seen thus far, if the current combination has more guests. Note that we cannot use the **max**-over-a-list trick in this optimized version since we did not store the entire list.

Applications

Our dinner problem is a classic one called the maximum independent set (MIS) problem: Given a graph with vertices and edges, find a maximum set of vertices that do not have edges between them. There are several situations where we might have to solve MIS. For example, suppose a franchise is trying to identify new locations such that no two locations are close enough to compete with each other. Construct a graph where the vertices are possible locations, and add edges between any two locations deemed close enough to interfere. The MIS gives you the maximum number of locations you can sell without cannibalizing sales.

MIS is a hard problem: Every known algorithm that guarantees the maximum cardinality of chosen candidates for all problem instances, including the algorithm we coded, can take time exponential in the number of guests for some instances. If someone came up with an efficient algorithm to produce the maximum selection in *all* independent set problem instances, whose runtime grows polynomially[4] in the number of guests, or was able to formally prove that no such algorithm exists, that person would have solved one of the remaining unsolved Millennium Prize Problems, would win a cash prize of $1 million, and —most importantly—would attain instant rock star status in computer science.

If we're not interested in the absolute maximum cardinality, we can use the greedy approach of repeatedly picking guests with the fewest dislikes to solve our problem in a way that is guaranteed to be fast, that is, only requiring a few scans over the graph.

Exercises

Puzzle Exercise 1: As do most people, you like some friends more than others. Say you can assign an integer weight to your affection, and the list of guests has not only the names, but also the weights, as shown below.

```
dislikePairs = [['Alice','Bob'], ['Bob','Eve']]
guestList = [('Alice', 2), ('Bob', 6), ('Cleo', 3),
                    ('Don', 10), ('Eve', 3)]
```

Modify the original and/or optimized code so it invites a collection of friends with the highest weight. In the above example, since you like Bob more than Alice and Eve combined, your guest list will change from the version without weights.

Exercise 2: One trouble with our solution is efficiency. We generate 2^n combinations for n possible guests. (For $n = 20$, this is over a million.) In our first example, we have two guests—Cleo and Don—who are not in any dislike relationships. The vertices corresponding to Cleo and Don have no edges connecting to them in your social circle graph. Cleo and Don are easygoing—they can be invited to dinner regardless of who else is invited. So we could have shrunk our guest list to 3 (dropping Cleo and Don), generated combinations, removed unfriendly combinations, and then found a combination with the maximum number of guests. In effect, we could run:

```
dislikePairs = [['Alice','Bob'], ['Bob','Eve']]
guestList = ['Alice', 'Bob', 'Eve']
```

```
InviteDinner(guestList, dislikePairs)
```

And obtain:

```
Optimum Solution: ['Alice', 'Eve']
```

Then we could add in Cleo and Don. This produces the same result, but we have only generated 2^3 combinations, not 2^5. Similarly, we could have dropped 'A' from our larger example with nine vertices and added it back later, since it does not have any edges connected to it.

Modify the original code, or the code you wrote for the weighted version of the puzzle, so it performs this optimization. That is, shrink the guest list by scanning dislike pairs and removing guests who are not in any of the pairs. Then add these guests into the invited combination.

Puzzle Exercise 3: Suppose you can deal with one pair of friends who dislike each other at dinner by placing them at opposite ends of the table, with you sitting in the middle. Modify the original and/or optimized code to invite the maximum number of friends to dinner, or find the maximum-weight group to invite. Remember, you cannot have two pairs of friends who dislike each other, even if one friend is common to both pairs.

For example, if you are given:

```
LargeDislikes = [['B', 'C'], ['C', 'D'], ['D', 'E'],
                 ['F', 'G'], ['F', 'H'], ['F', 'I'],
                 ['G', 'H']]
LargeGuestList = [('A', 2), ('B', 1), ('C', 3),
                  ('D', 2), ('E', 1), ('F', 4),
                  ('G', 2), ('H', 1), ('I', 3)]
```

And you run the code you have written:

```
InviteDinnerFlexible(LargeGuestList, LargeDislikes)
```

This should produce:

```
Optimum Solution: [('A', 2), ('C', 3), ('E', 1),
                   ('F', 4), ('I', 3)]
Weight is: 13
```

Note that F and I dislike each other and need to be placed at opposite ends of the table.

Notes

1. This title paraphrases a classic movie produced by Columbia Pictures, 1967.

2. This won't make a difference in terms of correctness—we will still find a maximum group of guests, regardless of order of guests, in the input list or in our representation of the combinations. We might return a different maximum-size grouping in case of ties, however.

3. Keep in mind that in Python 2.x, you will have to use **xrange** rather than **range** in the **for** loop that begins on line 3 to ensure that all `2**n` values for `i` are not stored.

4. The runtime grows as n^k, where n is the number of candidates, and k is a fixed constant.

9

America’s Got Talent

I have no particular talent. I am merely inquisitive.
—Albert Einstein

> Programming constructs and algorithmic paradigms covered in this puzzle: Representing two-dimensional tables using lists.

You have decided to produce a television show titled “Who’s Got Talent?”[1] Over spring break, you hold auditions for your show, after recruiting many candidates. Each candidate claims a certain set of talents (e.g., flower arranging, dancing, skateboarding), and you verify them during the audition. Most of the candidates don’t fulfill your expectations, but some do. Now you have a set of candidates who can each perform at least one talent.

In your show, you want to highlight a diverse set of talents. It won’t help ratings if you have different flower arrangements each week. You take your list of candidates and make a list of all their talents. Then you go to your producers for approval in including these talents on your show. Your producers whittle down the list (e.g., they don’t think regurgitation plays particularly well with audiences) and you settle on a final list. They also tell you to keep costs down.

You realize that the best way to keep costs down and ratings up is to contract out the fewest number of candidates while producing the most diverse show possible. You therefore want to select the *fewest* candidates who can perform *all* the talents on the final list you came up with.

Suppose you end up with the following candidates and talents.

Talent → Candidate ↓	Sing	Dance	Magic	Act	Flex	Code
Aly					✓	✓
Bob		✓	✓			
Cal	✓		✓			
Don	✓	✓				
Eve		✓		✓		✓
Fay				✓		✓

In the above example, you could select Aly, Bob, Don, and Eve, and cover all the talents. Aly covers Flex and Code, Bob covers Dance and Magic, Don covers Dance and Sing, and Eve covers Dance, Act, and Code. Together, they have all six talents.

Can you cover all the talents with fewer people? More generally, how can you select the fewest candidates (rows) who cover all the talents (columns) in any given table? Another example is shown below.

Talent → Candidate ↓	1	2	3	4	5	6	7	8	9
A				✓	✓		✓		
B	✓	✓						✓	
C		✓		✓		✓			✓
D			✓			✓			✓
E		✓	✓						✓
F							✓	✓	✓
G	✓		✓				✓		

In our first example, you *can* do better than four candidates. You only need to hire Aly, Cal, and Eve. Aly covers Flex and Code, Cal covers Sing and Magic, and Eve covers Dance, Act, and Code. Together, they have all six talents.

We will follow the same strategy as we did for the dinner invitation problem (puzzle 8). The two problems are quite similar, even though in the dinner puzzle we wanted to maximize the number of people, and here we want to minimize the number of selected people. What they have in common is that we have to look at

all possible combinations (i.e., subsets of candidates), eliminate subsets who do not collectively have all the talents, then choose the smallest subset. Another thing they have in common is that a greedy approach does not work.

The data structures of the two puzzles are different, since we need a table of information for our talent problem as opposed to a dislikes graph. Our example can be converted to data structures like this:

```
Talents = ['Sing', 'Dance', 'Magic', 'Act', 'Flex', 'Code']
Candidates = ['Aly', 'Bob', 'Cal', 'Don', 'Eve', 'Fay']
CandidateTalents = [['Flex', 'Code'], ['Dance', 'Magic'],
                    ['Sing', 'Magic'], ['Sing', 'Dance'],
                    ['Dance', 'Act', 'Code'], ['Act', 'Code']]
```

We have a list of talents (columns of the table) and a list of candidates (rows of the table). We need a list of lists, CandidateTalents, to represent the entries of the table. CandidateTalents has entries corresponding to rows of the table. The order of the entries is important because it reflects the order of the candidates in the list Candidates. Bob is the second candidate in Candidates and his talents correspond to the second list in CandidateTalents, namely, ['Dance', 'Magic'].

As you probably guessed, the code for the two puzzles is awfully similar. We'll structure the code for this puzzle slightly differently, and we will base it on the optimized version in puzzle 8.

Generating and Testing One Combination at a Time

Here's code for the top-level procedure, which generates each combination, tests whether it's good, and picks the minimum-length good combination.

```
1.  def Hire4Show(candList, candTalents, talentList):
2.      n = len(candList)
3.      hire = candList[:]
4.      for i in range(2**n):
5.          Combination = []
6.          num = i
7.          for j in range(n):
8.              if (num % 2 == 1):
9.                  Combination = [candList[n-1-j]] + Combination
10.             num = num // 2
11.         if Good(Combination, candList, candTalents, talentList):
12.             if len(hire) > len(Combination):
13.                 hire = Combination
```

```
14.          print ('Optimum Solution:', hire)
```

Lines 4–10 generate the combination corresponding to value num = i. Line 11 invokes a function Good—to be described later—which checks whether the given combination of candidates covers all the talents. If it does, it is compared against the best combination of candidates seen thus far, and the best combination is updated if it has a greater number of candidates (lines 12–13).

Line 3 is different from the equivalent line in puzzle 8, which was invite = []. In the dinner invitation puzzle, we wanted to maximize the number of guests we could invite, so we started with the empty list as the initial best combination. Here, we want to minimize the number of candidates we hire, so we start with the entire candidate list as the initial best combination. We are assuming that the entire set of candidates does satisfy our requirement of covering all the talents. If they don't, we can simply redefine our required talent list.

Determining Combinations That Lack a Talent

We invoked a function in Hire4Show that determines whether a given combination of candidates satisfies all the talents. The checks we need to make are quite different from those we applied in puzzle 8, as we will demonstrate below.

```
1.    def Good(Comb, candList, candTalents, AllTalents):
2.        for tal in AllTalents:
3.            cover = False
4.            for cand in Comb:
5.                candTal = candTalents[candList.index(cand)]
6.                if tal in candTal:
7.                    cover = True
8.            if not cover:
9.                return False
10.       return True
```

The **for** loop (lines 2–9) iterates over each talent tal in the list of talents. For each candidate in the combination (inner **for** loop starting on line 4), we use the candidate's index in the candidate list candList to index into the candidates-to-talents data structure (line 5).

We now have to check whether this candidate's talents contain the talent tal we are looking for in this iteration of the **for** loop (that begins on line 2). This is done on line 6. If the candidate has the talent tal, we mark it so on line 7. However, if

we complete the inner **for** loop and we have not found a candidate in the combination who covers talent tal, we know that this candidate combination needs to be discarded—one missing talent dooms a combination. Therefore, we need not bother checking for the other talents and we can simply return **False** (line 9).

If we get through all the talents without returning **False** in any of the iterations, it means that each talent is covered by this combination, and we can return **True** (line 10).

Let's run this code on the example from the beginning. The table, converted to code, is:

```
Talents = ['Sing', 'Dance', 'Magic', 'Act', 'Flex', 'Code']
Candidates = ['Aly', 'Bob', 'Cal', 'Don', 'Eve', 'Fay']
CandidateTalents = [['Flex', 'Code'], ['Dance', 'Magic'],
                    ['Sing', 'Magic'], ['Sing', 'Dance'],
                    ['Dance', 'Act', 'Code'], ['Act', 'Code']]
```

If we run:

```
Hire4Show(Candidates, CandidateTalents, Talents)
```

It produces:

```
Optimum Solution: ['Aly', 'Cal', 'Eve']
```

Exactly what we want and expect.

If we run the code on the second, larger example:

```
ShowTalent2 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
CandidateList2 = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
CandToTalents2 = [[4, 5, 7], [1, 2, 8], [2, 4, 6, 9],
                  [3, 6, 9], [2, 3, 9], [7, 8, 9],
                  [1, 3, 7]]
Hire4Show(CandidateList2, CandToTalents2, ShowTalent2)
```

It produces:

```
Optimum Solution: ['A', 'B', 'D']
```

Applications

This puzzle is an instance of the set-covering problem, which has numerous applications. For example, a car company may wish to deal with the smallest set of vendors who can supply all the parts it needs to build its cars, minimizing the amount of vetting that the company needs to do. Or NASA wishes to minimize

the total weight of the set of tools it sends into space, while ensuring that all maintenance tasks can be performed.

Set covering is a hard problem: Every known algorithm that guarantees the minimum cardinality of chosen candidates for all problem instances, including the algorithm we coded, can take time exponential in the number of candidates for some instances. In that sense, set covering is equivalent to our dinner puzzle (puzzle 8).

If we are not interested in the absolute minimum cardinality, we can use a greedy approach to solve our problem. It is guaranteed to be fast, that is, it will only require a few scans or passes over the table. A greedy algorithm for our puzzle would pick the candidate with the most talents. Once this candidate is picked, all talents covered by this candidate are eliminated from the table. The process continues until all talents are covered.

For our second, larger example with candidates *A* through *G*, we would first pick *C*, who covers four talents: 2, 4, 6, and 9. This results in the smaller table shown below.

Talent → Candidate ↓	1	3	5	7	8
A			✓	✓	
B	✓				✓
D		✓			
E		✓			
F				✓	✓
G	✓	✓		✓	

Here, candidate *G* covers three (additional) talents. *G* will be picked, since each of the others only covers two talents. After picking *C* and *G*, we obtain the table below.

Talent → Candidate ↓	5	8
A	✓	
B		✓
D		
E		
F		✓

We still need to cover talent 5, which requires candidate *A*, and talent 8, which requires candidate *B* or *F*. That is a total of four candidates. We know this is not optimal; our code gave us a three-candidate result.

Exercises

Exercise 1: In our first example, Eve "dominates" Fay since she has all the talents that Fay has, and more besides. Modify the code so candidates who are dominated are removed from the table. This will make the generation of combinations more efficient.

Exercise 2: Aly has to be selected in our first example because she is the only one who can Flex. You can see this clearly in the table, since the column for Flex only has a single checkmark. Similarly, in our second example, *D* is the only candidate who covers talent 4, and *F* the only candidate who covers talent 7.

Modify the original code to (1) identify candidates who each have a unique talent, (2) shrink the table to remove all talents covered by such candidates, (3) find the optimum selection for the shrunken table, and (4) add in the candidates you identified in step 1.

Puzzle Exercise 3: You think that some of the candidates are full of themselves and will demand to be paid more than what they deserve. You therefore want a way of choosing candidates who will settle for what you pay them. You assign a weight to each candidate that reflects what you think they might cost. Change the code so it finds a set of candidates that covers all the talents as before, but with minimum weight.

Suppose you are given:

```
ShowTalentW = [1, 2, 3, 4, 5, 6, 7, 8, 9]
CandidateListW = [('A', 3), ('B', 2), ('C', 1), ('D', 4),
                  ('E', 5), ('F', 2), ('G', 7)]
CandToTalentsW = [[1, 5], [1, 2, 8], [2, 3, 6, 9],
                  [4, 6, 8], [2, 3, 9], [7, 8, 9],
                  [1, 3, 5]]
```

where the numbers in the 2-tuples in CandidateListW correspond to how expensive each candidate is. Your code, which minimizes expense, should produce:

```
Optimum Solution: [('A', 3), ('C', 1), ('D', 4), ('F', 2)]
Weight is: 10
```

Note that the "domination" optimization is no longer valid, if Eve is going to be much more expensive than Fay! If Eve has the same weight as Fay or a smaller weight, only then is the optimization valid. So be careful if you are using code you wrote in exercise 1.

Exercise 4: Remember the optimization of essential candidates described in exercise 2? Add that to the code that solves the candidate selection problem for minimum weight, from puzzle exercise 3.

Note

1. This puzzle title is based off of NBC's talent show (2006–).

10

A Profusion of Queens

To loop is human, to recurse divine.
—Author unknown

Programming constructs and algorithmic paradigms covered in this puzzle: Recursive procedures. Exhaustive search through recursion.

Having solved the eight-queens problem (puzzle 4), we turn our attention to solving the *N*-queens problem for arbitrary *N*. That is, we need to place *N* queens on an $N \times N$ board such that no two queens attack each other.

Let's say we are not allowed to write nested **for** loops (or other types of loops) that have a degree of nesting more than 2. You might say this is an artificial constraint, but not only is the deeply nested code of puzzle 4 aesthetically displeasing, the code is also not general. If you wanted to write a program to solve the *N*-queens problem for *N* up to, say, twenty, you would have to write functions to solve four-queens (with four nested loops), five-queens (with five nested loops), all the way to twenty-queens (with twenty nested loops!), and invoke the appropriate function depending on the value of *N* when the code is run. What happens if you then want a solution to the twenty-one-queens problem?

We will need to use recursion to solve the general *N*-queens problem. Recursion occurs when something is defined in terms of itself. The most common use of recursion in programming is when a function is called within its

own definition.

In Python, a function can call itself. If a function calls itself, it is called a recursive function. Recursion may also mean a function A calling a function B, which in turn calls function A. We'll focus on the simple case of recursion here, namely a function f calling f again.

Recursive Greatest Common Divisor

What exactly happens when a function f calls itself? From an execution standpoint, surprisingly, this is not very different from the function f calling a different function g. Let's look at a simple case of recursion where we are computing the greatest common divisor (GCD) of a number. We can easily do this iteratively using the Euclidean algorithm, as shown below.

```
1.  def iGcd(m, n):
2.      while n > 0:
3.          m, n = n, m % n
4.      return m
```

Here's recursive code with equivalent functionality:

```
1.  def rGcd(m, n):
2.      if m % n == 0:
3.          return n
4.      else:
5.          gcd = rGcd(n, m % n)
6.          return gcd
```

We make two key observations:

. rGcd does not call itself in every case—there is a base case where if m % n == 0, then rGcd returns n and does not call itself. This corresponds to lines 2 and 3 above.

. The arguments to the rGcd invocation inside of rGcd—the callee rGcd on line 5—are different from the arguments of the caller rGcd. Over two recursive calls (i.e., rGcd calling rGcd calling rGcd), the arguments of the third call will be smaller than the arguments of the first.

Together these two observations ensure that rGcd terminates. If a function calls itself with exactly the same argument(s), then assuming there is no global state being modified and tested, we will end up with an infinite loop, that is, a

nonterminating program. We will also create a nonterminating program if we do not have a base case with no recursive call.

The execution of rGcd(2002, 1344) is shown in the following, illustrating the called procedures.

rGcd(2002, 1344) (line 5 call)
→ rGcd(1344, 658) (line 5 call)
→ rGcd(658, 28) (line 5 call)
→ rGcd (28, 14) (returns on line 3)
rGcd(658, 28) (returns on line 6)
rGcd(1344, 658) (returns on line 6)
rGcd(2002, 1344) (returns on line 6)

The indentation reflects the recursive calls.

Recursive Fibonacci

Switching to a different example, the first part of the famous Fibonacci sequence is:

0 1 1 2 3 5 8 13 21 34 55

In mathematical terms, a recurrence relation defines the sequence F_n of Fibonacci numbers:

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = 0, F_1 = 1$$

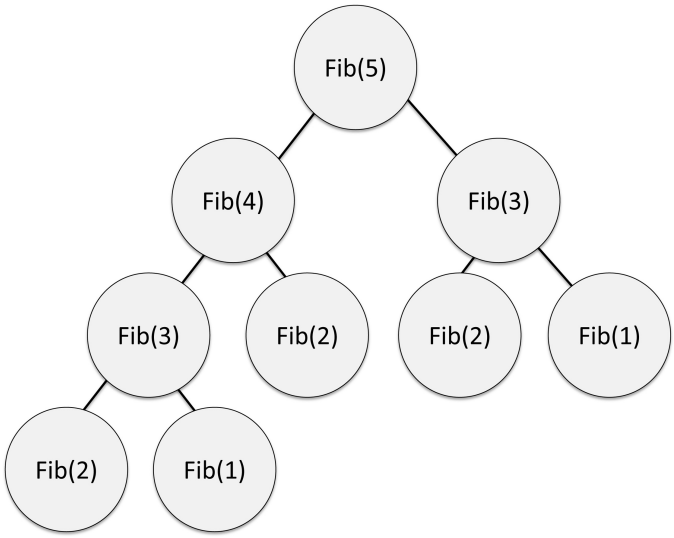
Fibonacci thus has a natural recursive definition, and the definition can easily be translated to the recursive code shown below.

```
1.  def rFib(x):
2.      if x == 0:
3.          return 0
4.      elif x == 1:
5.          return 1
6.      else:
7.          y = rFib(x-1) + rFib(x-2)
8.      return y
```

Notice how the code closely mirrors the recurrence. We have two base cases,

corresponding to lines 2–3 and lines 4–5. The recursive calls are made on line 7 —note the two recursive calls with different arguments. Further, we can make the same two observations about the recursive Fibonacci code that we made about the recursive factorial code, with respect to the base cases and the arguments of the recursive calls.

The execution of rFib(5) is shown below.



The execution above shows redundant work being performed. For example, rFib(3) is called twice. Of course it produces the same result of 2 each time. rFib(2) is called three times and returns 1 each time.

One way to produce efficient code that computes Fibonacci numbers with minimum computation is to use a different, iterative algorithm, as shown below.

```
1.  def iFib(x):
2.      if x < 2:
3.          return x
4.      else:
5.          f, g = 0, 1
6.          for i in range(x-1):
7.              f, g = g, f + g
8.          return g
```

Line 7 is the critical line where the next number in the sequence is computed by summing the previous two numbers, and the variables updated for the next iteration. The recursive code can be made as efficient as the iterative code above by remembering the result of an rFib(i) call in a table and looking up the table rather than making the same recursive call over and over. This technique, called

memoization, is the subject of puzzle 18.

Can you write code for a recursive algorithm that solves *N*-queens?

Recursive *N*-queens

The good news is that we do not have to throw away all the code we wrote for the eight-queens problem. We can retain the procedure noConflicts(board, current) that checks a partial configuration to see if it violates any of the three rules. The procedure is replicated below—it assumes a compact data structure with each column of the board being represented by a number. Specifically, −1 means no queen on the column, 0 means a queen on the first (topmost) row, and $n - 1$ means a queen on the bottommost row.

```
1.  def noConflicts(board, current):
2.      for i in range(current):
3.          if (board[i] == board[current]):
4.              return False
5.          if (current - i == abs(board[current] - board[i])):
6.              return False
7.      return True
```

Recall that this procedure only checks for conflicts between the newly placed queen on the column numbered current and previously placed queens on columns whose numbers are less than current. It does not check for conflicts between previously placed queens. Therefore, the recursive procedure that we write needs to call noConflicts each time it places a queen, just like EightQueens does. Finally, recall that the value of current can be less than the size of the board; the columns after current are empty.

These observations lead us to the recursive procedure below.

```
 1.  def rQueens(board, current, size):
 2.      if (current == size):
 3.          return True
 4.      else:
 5.          for i in range(size):
 6.              board[current] = i
 7.              if noConflicts(board, current):
 8.                  found = rQueens(board, current + 1, size)
 9.                  if found:
10.                      return True
```

11. **return False**

The recursive search corresponds to calling the procedure with more queens in fixed positions and fewer columns on which queens need to be placed. The first thing to look at in a recursive procedure is the base case. When does the recursion stop, that is, under what conditions are no more recursive calls made? Line 2 provides the base case: current has a value that is beyond the board. (The columns are numbered 0 to size - 1.)

The **for** loop on lines 5–10 goes through the choices in placing a queen on a particular column, numbered current. As we will show you later, the nQueens procedure that calls rQueens will call it with current = 0. Therefore, we can assume that the first time the **for** loop is entered, current = 0. A queen is placed on row i, where i varies from 0 to size - 1. i represents the row on which a queen will be placed on column current.

Obviously, the very first queen will not cause a conflict. That is, we know that noConflicts(board, 0) will return **True**, since the **for** loop in line 2 of noConflicts will have 0 iterations, given that we have **range**(0). However, placing the second or later queens might cause a conflict, and if it does, the procedure moves to the next row (next iteration of the **for** loop that begins on line 5). If not, we have found a partial solution with columns from 0 to current having queens that are not in conflict, and rQueens is recursively called with current + 1. If the recursive call returns **True**, then the caller returns **True** as well. If none of the calls corresponding to the queen placements on the row return **True**, the caller returns **False.**

Let's go back to the termination condition or base case, on lines 2 and 3. If current == size, it means that noConflicts(board, size - 1) is **True**. This is because we only call rQueens with current = j when noConflicts(board, j - 1) has returned **True**—the **if** statement on line 7 ensures this. If noConflicts(board, size - 1) is **True**, we have found a solution. This solution is contained in board since we have filled in size elements of board.

Now, look at nQueens below, which makes the first call to rQueens. It is a "wrapper" for the recursive procedure. The main reason we need such a procedure is because we need to initialize the board to be empty. If we initialized the board to be empty inside rQueens, each recursive call would empty the board. (Of course, there are ways to check if this is the first call to rQueens and initialize only in that case, but a clean way of handling initialization is to do it outside the recursive call.)

```
1.    def nQueens(N):
2.        board = [-1] * N
3.        rQueens(board, 0, N)
4.        print (board)
```

This procedure initializes the board to be empty (line 2) by creating a list with N elements, all of which are -1, then calls the recursive search procedure rQueens with an empty board and current = 0 (line 3), and prints the board (Line 4).

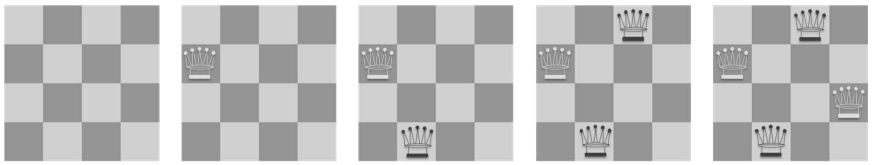
If you run:

```
nQueens(4)
```

You get:

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

This solution to the four-queens problem is discovered through the recursive calls illustrated below.



The arguments to rQueens—board, current, and size—are shown at the bottom of each board configuration. Each of these calls returns **True**. Failed calls are not shown—for example, the first recursive call made by rQueens([-1, -1, -1, -1], 0, 4) is to rQueens([0, -1, -1, -1], 1, 4), which returns **False** after many failed recursive calls.

If you run:

```
nQueens(20)
```

You get:

```
[0, 2, 4, 1, 3, 12, 14, 11, 17, 19, 16, 8, 15, 18, 7, 9, 6, 13,
5, 10]
```

Warning: The code's runtime grows exponentially with N, so if you run it with N >> 20 it will take a *very* long time!

Applications of Recursion

We used recursive enumeration in the *N*-queens problem. The enumeration

procedure checks for conflicts immediately after a queen is placed on the board. This is helpful in improving performance—we do not want to extend a configuration that already has a conflict. Adding queens to this configuration will not result in a valid solution.

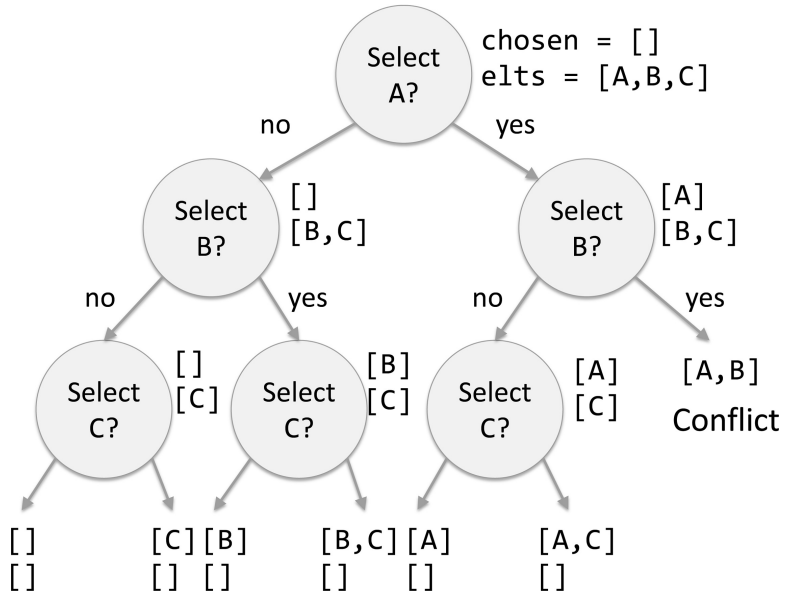
The way we solved the dinner and talent puzzles, puzzles 8 and 9, required enumerating all possible combinations of guests and candidates, respectively. The combinations correspond to each guest either being selected or not, and are subsets of the guest list. In the dinner puzzle, if Alice and Bob dislike each other, there is no sense in generating combinations such as [Alice, Bob, Eve], [Alice, Bob, Cleo], and so on. Now that we understand recursion, let's use it to solve the dinner problem in a more efficient way.

We will recursively generate combinations and perform early conflict detection so we don't extend bad combinations. We start with an empty combination and an available guest list that is initially equal to the list of guests. In our recursive strategy, we have two branches in the recursion:

. We add a new guest from the available guest list to the current combination, and recur only if the combination remains valid.
. We eliminate a guest from the available guest list without adding the guest to the current combination, and recur.

The base case is when the available guest list is empty. During the recursion, we have to maintain the best (i.e., maximum cardinality) solution found.

Suppose we have a guest list [A, B, C] and the dislike relationship [A, B]. Here's what the entire recursive execution (called the recursion tree) looks like. chosen corresponds to the current combination (i.e., the currently chosen guests), and elts corresponds to the guests available to be chosen.



The key observation here is that a conflict results in the termination of the branch. At the bottom we have the base cases, and the two maximum cardinality solutions [B, C] and [A, C]. One of these will be picked depending on the order that the tree is executed in, that is, whether the yes branch or the no branch is executed first.

The following is the function largestSol, which implements the recursion. It takes four arguments: chosen, elts, dPairs (which represents the dislike relationships), and Sol (which corresponds to the best solution found).

```
1.    def largestSol(chosen, elts, dPairs, Sol):
2.        if len(elts) == 0:
3.            if Sol == [] or len(chosen) > len(Sol):
4.                Sol = chosen
5.            return Sol
6.        if dinnerCheck(chosen + [elts[0]], dPairs):
7.            Sol = largestSol(chosen + [elts[0]],\
7a.                             elts[1:], dPairs, Sol)
8.        return largestSol(chosen, elts[1:], dPairs, Sol)
```

The base case (line 2) is that the list of available guests is empty. If Sol is empty, we have found our first solution (line 3) and we update Sol. If Sol is not empty, we check to see if we have found a larger solution (second part of line 3), and if so, we update Sol with the better solution. Sol is returned.

Lines 6–7 correspond to case 1 of the recursion. In the recursive step (line 6), we check if adding the first guest elts[0] in the available list to chosen results in a

conflict. If not, we add elts[0] to chosen, remove it from the available guest list by list slicing, and recur (lines 7 and 7a). Line 8 corresponds to case 2 of the recursion, where we recur without adding elts[0] to chosen.

The procedure dinnerCheck below should be familiar from puzzle 8.

```
1.    def dinnerCheck(invited, dislikePairs):
2.        good = True
3.        for j in dislikePairs:
4.            if j[0] in invited and j[1] in invited:
5.                good = False
6.        return good
```

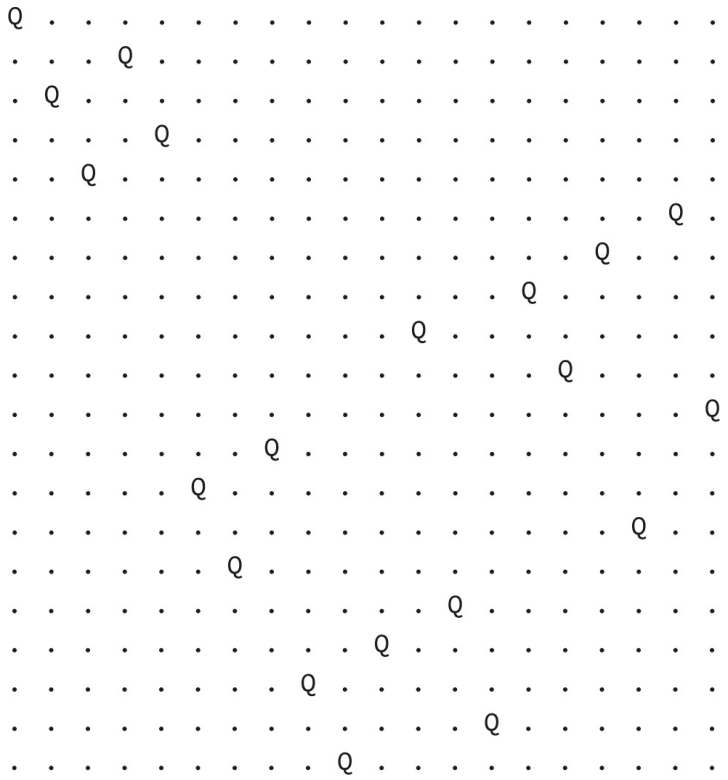
We go through the dislike relationships and check whether two guests who dislike each other are both in the invite list.

Finally, the procedure InviteDinner below calls largestSol with an empty invite list, the full guest list as the guests available to be chosen, the dislike relationships, and an empty solution list (line 2).

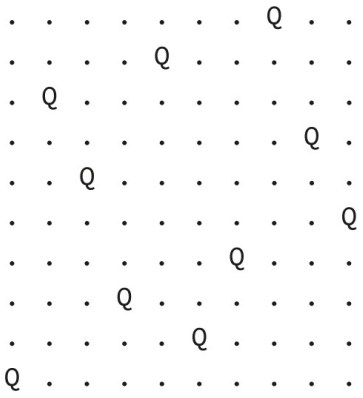
```
1.    def InviteDinner(guestList, dislikePairs):
2.        Sol = largestSol([], guestList, dislikePairs, [])
3.        print("Optimum solution:", Sol, "\n")
```

Exercises

Exercise 1: Modify the nQueens code to pretty-print a two-dimensional board, as shown below, with the solution obtained by nQueens(20). A period (.) signifies an empty square on the board, and a Q signifies a queen. There is a space between each pair of periods.



Puzzle Exercise 2: Modify the nQueens code so it looks for solutions with a queen already placed in a list of locations, and prints one if it exists. You can use a 1-D list location that has nonnegative entries for certain columns, which correspond to fixed queen positions. For example, location = [-1, -1, 4, -1, -1, -1, -1, 0, -1, 5] has three queens placed in the third, eighth, and tenth columns for a 10 × 10 board. Your code should produce the following solution, which is consistent with the specified locations.



Exercise 3: A palindrome is a string that that reads the same front-to-back and back-to-front. For example, “kayak” and “racecar” are palindromes. Write a

recursive function using list slicing that determines whether an argument string is a palindrome. Your procedure should ignore the case of letters (e.g., it should report that 'kayaK' is a palindrome).

Puzzle Exercise 4: Write a recursive algorithm to solve the "America's Got Talent" puzzle (puzzle 9). You can follow a structure similar to the recursive solution to the dinner puzzle, with key differences being that a solution to the talent puzzle, which is a combination of candidates, should ensure that all talents are covered, and the smallest solution should be returned. You should be able to reuse the function Good from puzzle 9, which determines if a selection of candidates covers all required talents.

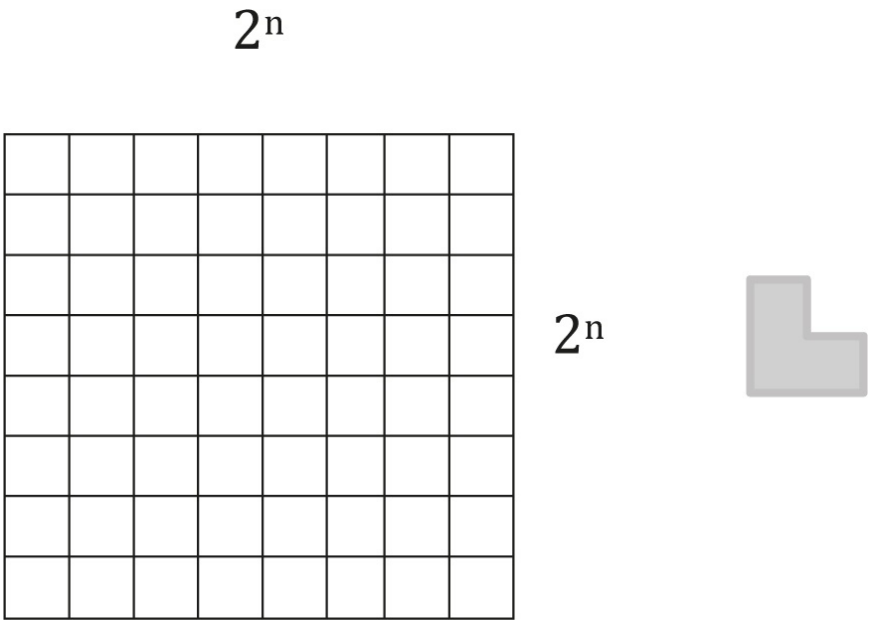
11

Tile That Courtyard, Please

We shape our buildings; thereafter they shape us.
—Winston Churchill

> Programming constructs and algorithmic paradigms covered in this puzzle: List comprehension basics. Recursive divide-and-conquer search.

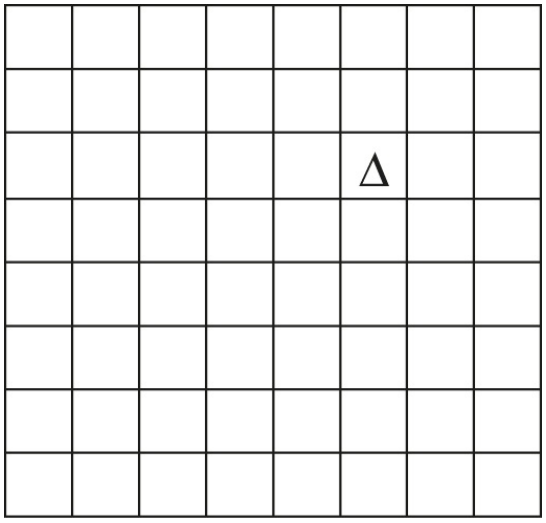
Consider the following tiling problem. We have a courtyard with $2^n \times 2^n$ squares and we need to tile it using L-shaped tiles, or trominoes. Each tromino consists of three square tiles attached to form an L shape, as shown below.



Can this be done without spilling over the boundaries, breaking a tromino, or having overlapping trominoes? The answer is no, simply because $2^n \times 2^n = 2^{2n}$ is not divisible by 3, only by 2. The fundamental theorem in arithmetic, namely the unique prime factorization theorem, states that every integer greater than 1 either is prime itself or is the product of prime numbers, and that this product is unique up to the order of the factors. Since 2 is prime, the theorem implies that 2^{2n} can *only* be written as a product of $2n$ 2's. Other primes, including 3, cannot be a factor of 2^{2n}. However, if there is one square that can be left untiled, then $2^{2n} - 1$ is divisible by 3. Can you show this?[1]

We therefore have hope of properly tiling a $2^n \times 2^n$ courtyard, leaving one square untiled because, for example, there is a statue of your favorite president on it. We'll call this square the missing square.

Is there an algorithm that tiles any $2^n \times 2^n$ courtyard with one missing square in an arbitrary location? As an example, below is a $2^3 \times 2^3$ courtyard where the missing square is marked Δ. Does the location of the missing square matter?

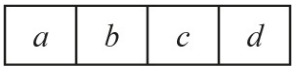


The answer is yes. We will describe and code a *recursive* divide-and-conquer algorithm that can tile a courtyard with $2^n \times 2^n$ squares, with one missing square in an arbitrary location. To help you understand how recursive divide-and-conquer works, we will first describe how it is used in merge sort, a popular sorting algorithm.

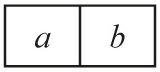
Merge Sort

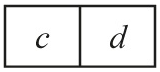
We can perform sorting using the elegant divide-and-conquer merge sort. Here's how merge sort works.

Suppose we have a list, as shown here with symbolic values or variables, that needs to be sorted in ascending order:

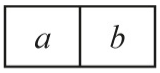


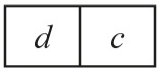
We divide it into two equal-sized[2] sublists:



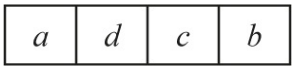


Then we sort the sublists recursively. At the base case of the recursion, if we see a list of size 2, we simply compare the two elements in the list and swap if necessary. Let's say that $a < b$ and $c > d$. Since we want to sort in ascending order, after the two recursive calls return we end up getting:





Now we come back to the first (or topmost) call to the sort function, and our task is to merge the two *sorted* sublists into one sorted list. We do this using a merge algorithm, which simply compares the first two elements of the two sublists, repeatedly. If $a < d$, it first puts a into the merged (output) list and effectively removes it from the sublist. It leaves d where it was. It then compares b with d. Let's say d is smaller. Merge puts d into the output list next to a. Next, b and c are compared. If c is smaller, c is placed next into the output list, and finally b is placed. The output will be:



Here's code for merge sort.

```
1.  def mergeSort(L):
2.      if len(L) == 2:
3.          if L[0] <= L[1]:
4.              return [L[0], L[1]]
5.          else:
6.              return [L[1], L[0]]
7.      else:
8.          middle = len(L)//2
9.          left = mergeSort(L[:middle])
10.         right = mergeSort(L[middle:])
11.         return merge(left, right)
```

Lines 2–6 are the base case, where we have a list of two elements and we place them in the right order. If the list has more than two elements, we split the list in two (line 8) and make two recursive calls, one on each sublist (lines 9–10). We use list slicing; `L[:middle]` returns the part of the list `L` corresponding to `L[0]` through `L[middle-1]`, and `L[middle:]` returns the part of the list corresponding to `L[middle]` to `L[len(L)-1]`, so no elements are dropped. Finally, on line 11, we call `merge` on the two sorted sublists and return the result.

All that remains is the code for `merge`.

```
1.    def merge(left, right):
2.        result = []
3.        i, j = 0, 0
4.        while i < len(left) and j < len(right):
5.            if left[i] < right[j]:
6.                result.append(left[i])
7.                i += 1
8             else:
9.                result.append(right[j])
10.               j += 1
11.       while i < len(left):
12.           result.append(left[i])
13.           i += 1
14.       while j < len(right):
15.           result.append(right[j])
16.           j += 1
17.       return result
```

Initially, merge creates a new empty list result (line 2). There are three **while** loops in merge. The first one, which is the most interesting, corresponds to the general case when the two sublists are both nonempty. In this case, we compare the current first elements of each of the sublists (represented by counters i and j), pick the smaller one, and increment the counter for the sublist whose element we selected to place in result. The first **while** loop terminates when either of the sublists becomes empty.

When one of the sublists is empty, we simply append the remaining elements of the nonempty sublist to the result. The second and third **while** loops correspond to the left sublist being nonempty and the right one being nonempty, respectively.

Merge Sort Execution and Analysis

Suppose we have this input list for merge sort:

```
inp = [23, 3, 45, 7, 6, 11, 14, 12]
```

How does execution proceed? The list is split in two:

```
[23, 3, 45, 7]          [6, 11, 14, 12]
```

and the left list is sorted first, and the first step is to split it in two:

```
[23, 3]          [45, 7]          [6, 11, 14, 12]
```

Each of these two-element lists is sorted in ascending order:

```
[3, 23]          [7, 45]          [6, 11, 14, 12]
```

The sorted lists of two elements each are merged into a sorted result:

```
[3, 7, 23, 45]                    [6, 11, 14, 12]
```

Next, the algorithm looks at the right list and splits it in two:

```
[3, 7, 23, 45]                    [6, 11]          [14, 12]
```

Each of the right sublists is sorted:

```
[3, 7, 23, 45]    [6, 11]          [12, 14]
```

The sorted sublists of two elements each are merged into a sorted result:

```
[3, 7, 23, 45]                    [6, 11, 12, 14]
```

And finally, the two sorted sublists of four elements each are merged:

```
[3, 6, 7, 11, 12, 14, 23, 45]
```

Merge sort is a more efficient sorting algorithm than the selection sort algorithm we described in puzzle 2. Selection sort had a doubly nested loop and therefore required, in the worst case, n^2 comparisons and exchanges for a list of length n. In merge sort, we only perform n operations in each merge step of a list of length n. The top-level merge will require n operations, and the next level will run `merge` on two lists of length $n/2$, and will also require n operations. The next level after that will require $n/4$ operations on each of four lists, for a total of n operations. The number of levels is $\log_2 n$, so merge sort requires $n \log_2 n$ operations.

As you can see, during the `merge` step we need to create a new list `result` (line 2) and return that. The amount of intermediate storage required for merge sort therefore grows with the length of the list to be sorted. This was not the case for selection sort. We will come back to this requirement in puzzle 13.

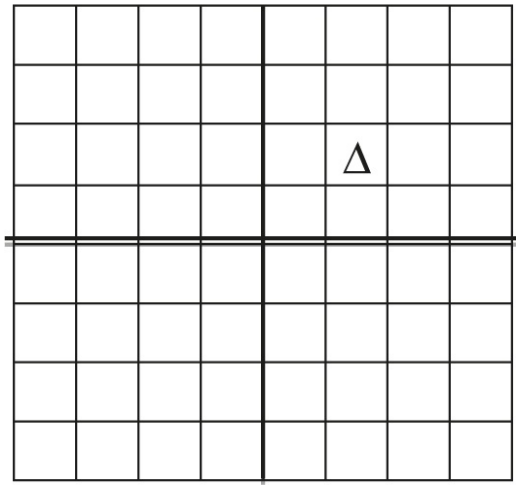
Now, let’s get back to courtyard tiling.

Base Case of a 2 × 2 Courtyard

Let’s start with $n = 1$, that is, a 2 × 2 courtyard with one missing square. The missing square can be in different locations, as shown below, where the Δ’s are

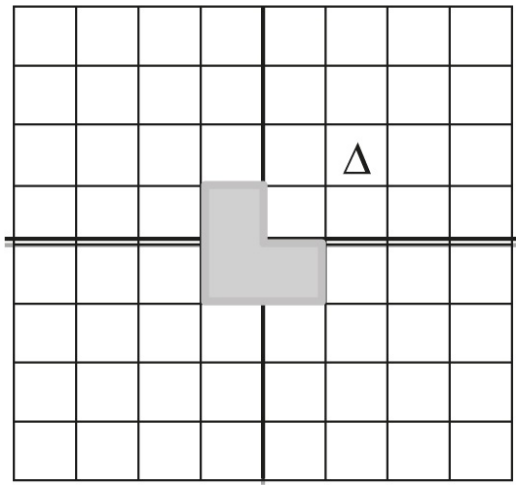
in different locations. In all four of these cases, we can tile the remaining three squares with an L-shaped tile oriented appropriately.

This is an important step because we now have a base case for a recursive divide-and-conquer algorithm that we can apply. But how can we divide the $2^n \times 2^n$ courtyard with one missing square, so that we get subproblems that are the same problem but smaller? If we divide the $2^n \times 2^n$ courtyard into four $2^{n-1} \times 2^{n-1}$ courtyards as shown below, we get one $2^{n-1} \times 2^{n-1}$ courtyard with a missing square—the one on the top right—but the other three are full $2^{n-1} \times 2^{n-1}$ courtyards.



Recursive Step

In the above example, we recognize that the statue Δ is in the top right quadrant. Therefore we place a tromino strategically on the three “full” quadrants to create a set of four quadrants, *each* with one missing square that needs to be tiled (see the following example).

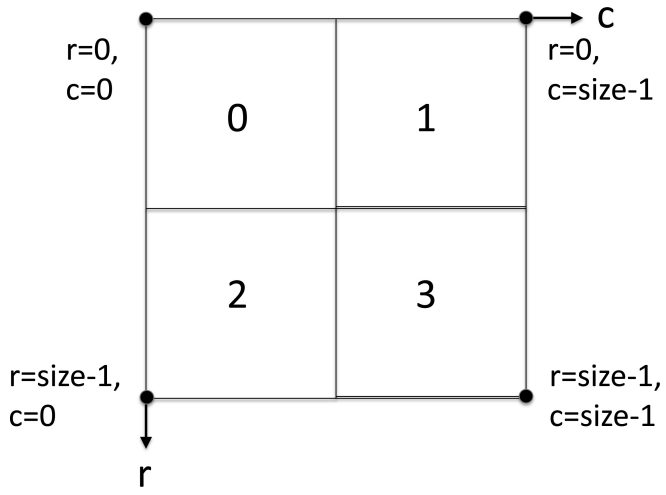


The top right quadrant is unchanged, but the other three have exactly one square tiled, so the remaining work is to tile four $2^{n-1} \times 2^{n-1}$ courtyards with one missing square each. Sound familiar? Note that if the missing square were in the top left quadrant, we would rotate the tromino counterclockwise 90 degrees and obtain four smaller courtyards as before. Work out the rotations for the other two cases.

We do this recursively till we obtain 2×2 courtyards with one missing square. Each of these is trivially tiled with one tromino, as we showed earlier. We're done.

Except not quite. You have a large $2^n \times 2^n$ courtyard (with a statue) to tile, and working with L-shaped tiles is a complicated job. This means you have to be very specific to the flooring contractor about each tile that needs to be laid down. We need to write a program that will generate a "map" of all the tiles for this courtyard. (This is a book about programming—you didn't think we'd stop with the algorithm, did you?)

We are going to number quadrants in a particular way, as shown in the code we write, below. Our courtyard will be represented as `yard[r][c]`, where `r` is the row number and `c` is the column number. Row numbers increase from top to bottom, and column numbers increase from left to right, as shown below.




```
1.  def recursiveTile(yard, size, originR, originC, rMiss,\
1a.                                                 cMiss, nextPiece):
2.          quadMiss = 2*(rMiss >= size//2) + (cMiss >= size//2)
3.          if size == 2:
4.                  piecePos = [(0,0), (0,1), (1,0), (1,1)]
5.                  piecePos.pop(quadMiss)
6.                  for (r, c) in piecePos:
7.                          yard[originR + r][originC + c] = nextPiece
8.                  nextPiece = nextPiece + 1
9.                  return nextPiece
10.         for quad in range(4):
11.                 shiftR = size//2 * (quad >= 2)
12.                 shiftC = size//2 * (quad % 2 == 1)
13.                 if quad == quadMiss:
14.                         nextPiece = recursiveTile(yard, size//2,\ originR +\
14a.                                shiftR, originC + shiftC, rMiss\ - shiftR,\
14b.                                cMiss - shiftC, nextPiece)
15.                 else:
16.                         newrMiss = (size//2 - 1) * (quad < 2)
17.                         newcMiss = (size//2 - 1) * (quad % 2 == 0)
18.                         nextPiece = recursiveTile(yard, size//2,\ originR +\
18a.                                shiftR, originC + shiftC, newrMiss,\ newcMiss,\
18b.                                nextPiece)
19.         centerPos = [(r + size//2 - 1, c + size//2 - 1)
19a.                                for (r,c) in [(0,0), (0,1), (1,0), (1,1)]]
20.         centerPos.pop(quadMiss)
21.         for (r,c) in centerPos:
22.                 yard[originR + r][originC + c] = nextPiece
23.         nextPiece = nextPiece + 1
24.         return nextPiece
```

The function recursiveTile takes as arguments a two-dimensional grid yard, the dimension of the current yard or quadrant being tiled (size), the row and column coordinates of the origin (originR, originC), and the location of the missing square (rMiss, cMiss) in relation to the origin. As a final argument, it also takes a helper variable, nextPiece, which is used to number tiles so we can print an easy-to-read map for the contractor. Assume for now that the origin is (0, 0).

The quadrant with the missing square is identified based on the location of the missing square on line 2, shown below:

```
quadMiss = 2*(rMiss >= size//2) + (cMiss >= size//2)
```

Rows are numbered from 0 at the top to size - 1 at the bottom. Columns are numbered 0 at the left to size - 1 on the right. As an example, if rMiss = 0, and cMiss = size - 1, we have the missing square in the top right quadrant, and quadMiss is computed as 2 * 0 + 1 = 1. For large rMiss and cMiss equal to or greater than size//2, we will get the lower right quadrant, numbered 3.

The base case is written first in recursiveTile, lines 3–9, and shows how nextPiece is used to mark the squares with the number of the tile that was used to cover the squares. The base case is for a 2 × 2 courtyard with one missing square in square quadMiss. We know we can tile the courtyard regardless of what quadMiss is, and we number the tile using nextPiece and fill in the courtyard yard. Since we do not want to tile the square quadMiss (it has either been tiled already or will have a statue on it), we remove it from the list of tuples piecePos using the pop function on the list. The pop function takes the index of the element in the list as an argument and removes that element from the list (line 5). Since the same data structure yard is filled in each recursive call, we need to use the origin coordinates to tile different, disjoint quadrants in each call (line 7). Once the 2 × 2 courtyard has been tiled, we increment nextPiece and return it (lines 8–9).

The overall recursion works as follows. Based on quadMiss, four recursive calls are made (lines 10–18). quadMiss is the location of the missing square in the yard, and the quadrant corresponding to quadMiss will have the missing square. However, in the three other quadrants, we will also have a missing square at one of the corners, since we will eventually place a tile at those corners. This center tile is placed after the recursive calls return (lines 19–23).

There is some computation to determine the arguments for the recursive calls. The size of the courtyard is size//2 and we pass nextPiece into the recursive calls. We also need to compute the origin coordinates for each of the four quadrants. The shifts required in the coordinates are computed in lines 11–12. Given our

numbering of the quadrants, when we make the recursive calls, quadrants 0 and 1 will have the same origin row coordinates as in the parent procedure, and quadrants 2 and 3 will have their origin row coordinates shifted by size//2 in relation to the parent procedure (line 11). Quadrants 0 and 2 will have the same origin column coordinates as the parent procedure, and quadrants 1 and 3 will have origin column coordinates shifted by size//2 (line 12).

For the recursive call corresponding to the quadrant quadMiss, which has the square that was missing in the parent call, we simply compute the new rMiss and cMiss in relation to the shifted origin coordinates (line 14).

The computation of the rMiss and cMiss arguments is different for the other three recursive calls because the missing square will be at one of the corners and is unrelated to the values of rMiss and cMiss in the parent call. The computation is given in lines 16–17. For quadrant 0, the bottom right corner will be the missing square, and its coordinates in relation to the origin are rMiss, equaling size//2 - 1, and cMiss, equaling size//2 - 1. The other quadrants work similarly.

Finally, in lines 19–23, we place the center tile in the yard, again based on quadMiss, which points to the square on which the tile should *not* be placed. We could just as easily have placed the center tile before making the recursive calls. The only difference would be in the numbering of the tiles, not where the tiles are placed. Lines 19 and 19a show the creation of a list centerPos in Python list comprehension style—more on that below. This list initially contains the four middle squares in the courtyard being processed in this call, which are each corner squares of the four quadrants of the courtyard. Line 20 removes from centerPos the corner square that belongs to the quadrant containing quadMiss.

Let's look at how to invoke recursiveTile.

```
1.    EMPTYPIECE = -1

2.    def tileMissingYard(n, rMiss, cMiss):
3.        yard = [[EMPTYPIECE for i in range(2**n)]
3a.                for j in range(2**n)]
4.        recursiveTile(yard, 2**n, 0, 0, rMiss, cMiss, 0)
5.        return yard
```

We will use -1 to signify an empty square in the yard (line 1). The function tileMissingYard is simply a wrapper for the function recursiveTile, which needs the arguments corresponding to the origin coordinates and nextPiece all initialized to 0. On lines 3 and 3a, we create a new two-dimensional list corresponding to a yard with each dimension equaling 2**n. We initialize it in Python list

comprehension style, which is more compact than the standard nested **for** loops we would need to fill a two-dimensional list. It's worth emphasizing that we allocate no memory for courtyards within the recursiveTile procedure—each recursive call fills in disjoint parts of the variable yard that is passed into the procedure.

The returned nextPiece value is ignored for the top-level call to recursiveTile, but is used in subsequent recursive calls.

List Comprehension Basics

List comprehensions can be used to create lists in a natural way. Suppose we want to produce the lists *S* and *O*, defined mathematically as:

$S = \{x^3 : x \text{ in } \{0 \ldots 9\}\}$
$O = \{x \mid x \text{ in } S \text{ and } x \text{ odd}\}$

Here's how we can produce the lists using list comprehensions:

```
S = [x**3 for x in range(10)]
O = [x for x in S if x % 2 == 1]
```

The first expression in the list definition corresponds to an element of the list, and the rest generates list elements according to specified properties. We are including in S the cubes of all numbers from 0 to 9. We are including all odd numbers in S in the list O.

Here's a more interesting example where we compute the list of prime numbers less than 50. First, we build a list of composite (non-prime) numbers, using a single list comprehension, and then we use another list comprehension to get the "inverse" of the list, which comprises the prime numbers.

```
cp = [j for i in range(2, 8) for j in range(i*2, 50, i)]
primes = [x for x in range(2, 50) if x not in cp]
```

The two loops in the definition of cp find all the multiples of numbers from 2 to 7 that are less than 50. The number 7 was chosen because $7^2 = 49$, the highest number less than 50. Some composite numbers may be repeated in the cp list. The definition for primes simply walks through numbers from 2 to 49 and includes numbers that are *not* present in the cp list.

List comprehensions can produce very compact code, which is sometimes incomprehensible. Use them in moderation!

Pretty Printing

One reason for this coding exercise was to produce a contractor-friendly map, and here is the printing routine that does exactly that.

```
1.  def printYard(yard):
2.      for i in range(len(yard)):
3.          row = ''
4.          for j in range(len(yard[0])):
5.              if yard[i][j] != EMPTYPIECE:
6.                  row += chr((yard[i][j] % 26) + ord('A'))
7.              else:
8.                  row += ' '
9.          print (row)
```

The procedure prints the two-dimensional `yard` on one line per row. It creates a row of characters corresponding to the tiles on that row, and prints the row. There is one missing square in the courtyard that is left empty by printing a space (line 8).

We could print the numbers, but we chose to replace them with letters `A` through `Z`.

The function `chr` takes a number and produces a letter associated with that number in ASCII format. The function `ord` is the inverse of `chr`—it takes a letter and produces its ASCII number. The tile with number `0` is assigned the letter `A` on line 5, and the tile with number `1` is assigned the letter `B`, and so on till number `26` gets `Z`. If the courtyard is $2^5 \times 2^5$ or larger, we will have more than 26 L-shaped tiles, and some tiles will get the same letter. (Of course, we could have printed these tiles' unique numbers. One problem with printing numbers is that we would have to deal with single-digit and double-digit numbers during printing.)

Let's run:

```
printYard(tileMissingYard(3, 4, 6))
```

to get:

```
AABBFFGG
AEEBFJJG
CEDDHHJI
CCDUUHII
KKLUPP Q
KOLLPTQQ
MOONRTTS
```

This is descriptive enough for any contractor to tile the courtyard properly. You can see in what order the tiles are laid by recursiveTile. The recursive calls follow the order of the quadrants: 0, 1, 2, and 3 (line 12 of recursiveTile). The first tile laid is tile A, which corresponds to the top left quadrant. The center tile, U, is laid last since we chose to lay down the center tile after the recursive calls returned. If we had laid the center tile before making the recursive calls, the center tile would have been A.

Let's do a runtime analysis of recursiveTile. It runs fairly fast on large yards. The key observation is that the yard size (i.e., the length and width of the courtyard) in each recursive call is *half* of the original. So if we start with a $2^n \times 2^n$ courtyard, in $n - 1$ steps we will get to the base case: a 2×2 courtyard. Of course, at each step, we are making four recursive calls, giving us 4^{n-1} calls, each of which tiles a 2×2 courtyard. The number of squares processed is exactly $2^n \times 2^n$, which is the number of squares in the initial courtyard. Of course, one of the squares is not tiled.

A Different Tiling Puzzle

The famous mutilated chessboard tiling puzzle goes as follows: Suppose a standard 8×8 chessboard has two diagonally opposite corners removed, leaving 62 squares. Is it possible to place 31 dominoes of size 2×1 so as to cover all these squares?

Exercises

Exercise 1: You are given a $2^n \times 2^n$ courtyard to tile with L-shaped tiles with four tiles missing. There are (at least) two cases where this is possible:

. When the four missing tiles are in four different quadrants.
. When any three of the four tiles are such that you can tile them using a tromino.

Write a procedure that will determine whether you can tile the courtyard using recursiveTile by checking for the above two conditions. The procedure can just return **True** or **False**, given n and a four-element list of missing square coordinates.

Puzzle Exercise 2: Suppose you have a two-dimensional list or matrix T as

shown below, where all rows and all columns are sorted. Devise and implement a binary search algorithm that works for lists such as T. You can assume that all elements are unique, as in the example below.

```
T = [[1, 4, 7, 11, 15],
     [2, 5, 8, 12, 19],
     [3, 6, 9, 16, 22],
     [10, 13, 14, 17, 24],
     [18, 21, 23, 26, 30]]
```

Here's the strategy: Guess that the value is at position i, j and think of what it means if the value is less than T[i][j], or if the value is greater than T[i][j]. For example, if we are searching for 21 and we compare with T[2][2] = 9, we know that 21 cannot be in the T[<= 2][<= 2] upper left quadrant because all values in that quadrant are less than 9. However, 21 could be in any of the three other quadrants: the bottom left T[> 2][<= 2] quadrant, which it is in this example, or the top right T[<= 2][> 2] or bottom right T[> 2][> 2] quadrants, in different examples.

You can always eliminate one of the four quadrants in your two-dimensional binary search. You will have to make recursive calls on the other three quadrants.

Puzzle Exercise 3: It is natural to ask whether the two-dimensional binary search algorithm of exercise 2 is the best possible algorithm. Let's look at T again:

```
T = [[1, 4, 7, 11, 15],
     [2, 5, 8, 12, 19],
     [3, 6, 9, 16, 22],
     [10, 13, 14, 17, 24],
     [18, 21, 23, 26, 30]]
```

Suppose we want to find out if element 13 exists in T. Here's the strategy: Start with the top right element. If the element is smaller than 13, you can eliminate the entire first row and move one position down. If the element is larger, you can eliminate the entire last column and move one position left. Obviously, if the element is the top right element, you can stop.

The neat thing about this strategy is that it eliminates either a row or a column in each step. So you will find the element you are searching for in at most $2n$ steps for an $n \times n$ matrix, or you will determine that it does not exist. Code the above algorithm by making recursive calls on the appropriate submatrix (with one less row or one less column). In our example above, we will move from 15 to 11 to 12 to 16 to 9 to 14 to 13.

Notes

1. Since this is not a book of mathematical puzzles, we'll tell you! $2^{2n} - 1$ can be written as $(2^n - 1)(2^n + 1)$. And 2^n is not divisible by 3, as we showed earlier. That means that either $2^n - 1$ is divisible by 3, or $2^n + 1$ is divisible by 3.

2. If the number of elements is odd, the subarrays are not equal in size, but differ only by 1.

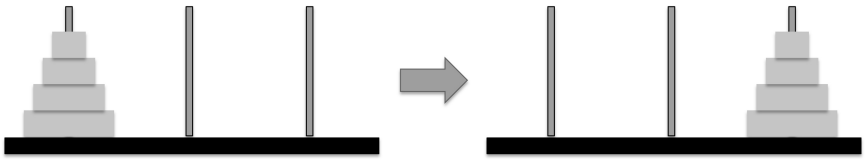
12

The Towers of Brahma with a Twist

Don’t wake me for the end of the world unless it has very good special effects.
—Roger Zelazny, *Prince of Chaos*

> Programming constructs and algorithmic paradigms covered in this puzzle: Recursive decrease-by-one search.

The Towers of Brahma, more popularly known as the Towers of Hanoi, is a mathematical game or puzzle. It consists of three pegs and a number of rings of different sizes, which can slide onto any peg. The puzzle starts with the rings in a neat stack on one peg, in order of size—the smallest at the top, the largest at the bottom—making a conical shape, as shown below.



The objective of the Towers of Hanoi (TOH) puzzle is to move the entire stack to another peg, obeying these simple rules:

Only one ring can be moved at a time.

Each move consists of taking the upper ring from one of the stacks and placing it on top of another stack (i.e., a ring can only be moved if it is the uppermost

ring on a stack).

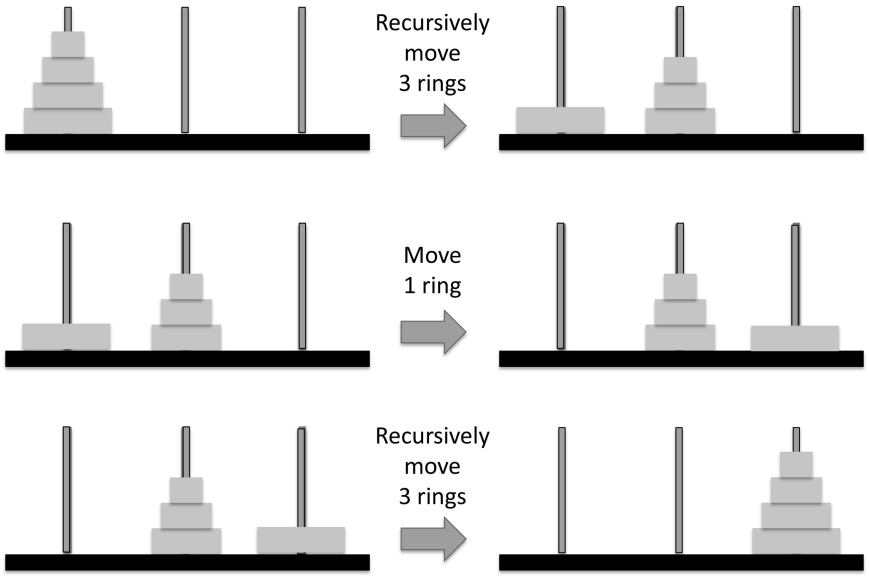
No ring may be placed on top of a smaller ring.

The puzzle was probably invented by the French mathematician Édouard Lucas in 1883—he certainly popularized it. Legend has it that the Hindu temple Kashi Vishwanath contained a large room with three posts in it, one of which initially had sixty-four golden rings. Since the dawn of time, Brahmin priests have been moving these rings from post to post following the rules above. According to the legend, the world will end when the last move of the puzzle is completed, that is, all the rings are properly placed on the destination post. It is not known if Lucas himself invented this legend or was inspired by it.

The original puzzle had sixty-four rings and three pegs; we will parameterize the puzzle to have *n* rings. There are many variants of the puzzle with differing numbers of pegs. We will first look at two versions of the puzzle, the classic one with three pegs, and a variant, which also has three pegs but has additional constraints on how rings can be moved. The exercises at the end of the chapter deal with yet more variants.

Recursive Solution of TOH

TOH can be solved using the algorithmic paradigm of divide and conquer. This approach is shown below, for the case where we wish to move the rings from the leftmost (start) peg to the rightmost (end) peg.



The first step (arrow) means a recursive call for a problem with $n - 1$ rings (the

example above has $n = 4$), with the same start peg but a different end peg. Since all three pegs are equivalent in TOH, it doesn't really matter what the start, middle, and end pegs are. The second step is moving one ring from the start peg to the end peg. And the third step means making another recursive call for a problem of size $n - 1$, with the start peg being the middle peg, and the end peg being the original end peg.

Note that this divide-and-conquer algorithm makes recursive calls where the problem size is one ring less than the original problem. The tiling puzzle (puzzle 11) made recursive calls corresponding to courtyard problems that were one-fourth the area.

We know how to take divide-and-conquer algorithms and turn them into recursive code. Here's a TOH recursive implementation that prints out each move required to solve TOH for numRings rings so the Brahmin priests can follow the immutable rules described earlier.

```
1.  def hanoi(numRings, startPeg, endPeg):
2.      numMoves = 0
3.      if numRings > 0:
4.          numMoves += hanoi(numRings - 1, startPeg,
                                          6 - startPeg - endPeg)
5.          print ('Move ring', numRings, 'from peg',
                                  startPeg, 'to peg', endPeg)
6.          numMoves += 1
7.          numMoves += hanoi(numRings - 1,
                              6 - startPeg - endPeg, endPeg)
8.      return numMoves
```

First, note that the rings are numbered from 1 through numRings in this code, beginning with the top ring (1) to the bottom ring (numRings). Next, pay attention to how the code is written to work with moving rings from any start peg to any end peg. The pegs are numbered 1, 2, and 3 from left to right. Given a start peg number and an end peg number, we can derive the other peg as 6 - startPeg - endPeg. This will be important since the recursive calls move rings between different pairs of pegs.

Lines 4, 5, and 7 correspond to the three arrows in the picture. Line 4—a recursive call—solves the problem of moving the top numRings - 1 rings from the start peg to the middle peg. We are counting the number of moves using the variable numMoves. Line 5 simply prints the single move required to move the bottommost ring, numbered numRings, from the start peg to the end peg. Line 7 makes a recursive call to move numRings - 1 rings from the middle peg to the end

peg.

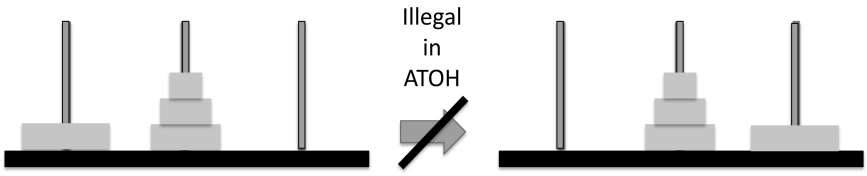
Let's run:

```
hanoi(3, 1, 3)
```

We have three rings, and `startPeg = 1` and `endPeg = 3`. This will call `hanoi(2, 1, 2)`, and will then move ring `3` from peg `1` to peg `3`, and will finally call `hanoi(2, 2, 3)`. The two recursive calls will each make two calls, each of which solves a Hanoi problem with one ring. We therefore output:

```
Move ring 1 from peg 1 to peg 3
Move ring 2 from peg 1 to peg 2
Move ring 1 from peg 3 to peg 2
Move ring 3 from peg 1 to peg 3
Move ring 1 from peg 2 to peg 1
Move ring 2 from peg 2 to peg 3
Move ring 1 from peg 1 to peg 3
```

Let's now look at a variant problem. In the Adjacent Towers of Hanoi (ATOH), we are not allowed to move rings between the leftmost and rightmost pegs, but only between adjacent pegs. In our recursive TOH algorithm, the second step (shown below), moving the single ring from the leftmost to rightmost peg, would be illegal.

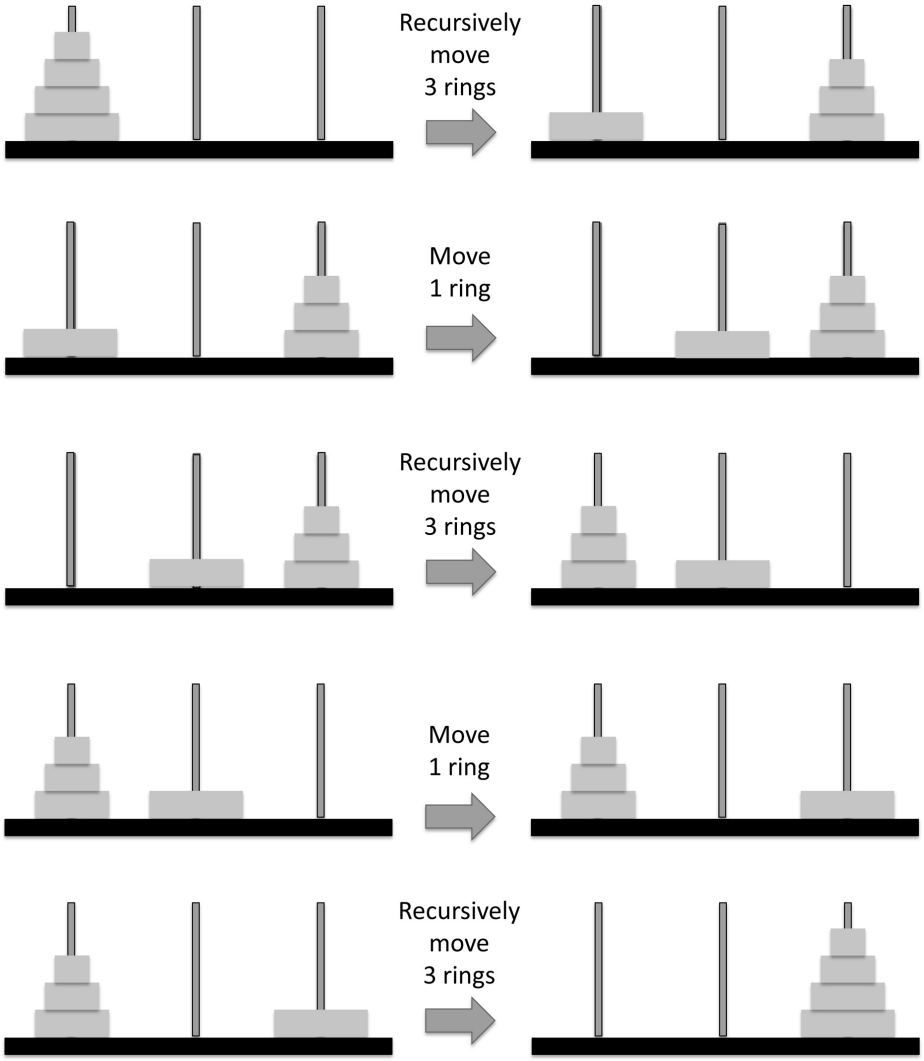


But we cannot move that ring to the middle peg, since a bigger ring cannot be on top of a smaller one. So we need a different recursive strategy to solve ATOH.

Can you think of a recursive divide-and-conquer strategy to solve ATOH?

Recursive Solution of ATOH

The key observation in divide-and-conquer is that as long as you have a base case, you can "pretend" that you know how to solve a smaller recursive problem. Clearly, in the case of one ring (i.e., $n = 1$), we can solve ATOH in two moves, by moving the ring from the leftmost peg to the middle peg, and then the rightmost peg. Let us then assume that we know how to solve ATOH for $n - 1$ rings. The recursive divide-and-conquer strategy is shown in the following diagram.



Note that the first recursive call assumes that ATOH can be solved for $n - 1$ rings, obeying the ATOH constraints. The start peg and the end peg are the same as for the original problem. Next, we have a single move that is legal in ATOH —moving the bottommost ring from the start peg to the middle peg. Then we make another recursive call to solve a problem with $n - 1$ rings, with the roles of the start and end pegs from the original problem interchanged. The start and end pegs are symmetric in ATOH, and their roles are interchangeable. Next we move the largest ring from the middle peg to the end peg. And finally, we solve a problem of $n - 1$ rings corresponding to the same start and end pegs as the original ATOH problem with n rings.

Let's look at the ATOH recursive implementation.

```
1.    def aHanoi(numRings, startPeg, endPeg):
2.        numMoves = 0
```

```
3.          if numRings == 1 :
4.              print ('Move ring', numRings, 'from peg',
                        startPeg, 'to peg', 6 - startPeg - endPeg)
5.              print ('Move ring', numRings, 'from peg',
                            6 - startPeg - endPeg, 'to peg', endPeg)
6.              numMoves += 2
7.          else:
8.              numMoves += aHanoi(numRings - 1, startPeg, endPeg)
9.              print ('Move ring', numRings, 'from peg', startPeg,
                                              'to peg', 6 - startPeg - endPeg)
10.             numMoves += 1
11.             numMoves += aHanoi(numRings - 1, endPeg, startPeg)
12.             print ('Move ring', numRings, 'from peg',
                            6 - startPeg - endPeg, 'to peg', endPeg)
13.             numMoves += 1
14.             numMoves += aHanoi(numRings - 1, startPeg, endPeg)

15.         return numMoves
```

This code has more steps than the code for TOH but is not conceptually more complicated. As before, the middle peg can be derived from numeric values of startPeg and endPeg. The base case of one ring requires two moves (lines 4 and 5). The code for numRings > 1 follows the steps in the picture, making three recursive calls and two single ring moves. For example, the second recursive call on line 11 flips the roles of the original start peg and end peg, as the picture indicates.

Let's run:

```
aHanoi(3, 1, 3)
```

This outputs:

```
Move ring 1 from peg 1 to peg 2
Move ring 1 from peg 2 to peg 3
Move ring 2 from peg 1 to peg 2
Move ring 1 from peg 3 to peg 2
Move ring 1 from peg 2 to peg 1
Move ring 2 from peg 2 to peg 3
Move ring 1 from peg 1 to peg 2
Move ring 1 from peg 2 to peg 3
Move ring 3 from peg 1 to peg 2
Move ring 1 from peg 3 to peg 2
Move ring 1 from peg 2 to peg 1
Move ring 2 from peg 3 to peg 2
Move ring 1 from peg 1 to peg 2
```

```
Move ring 1 from peg 2 to peg 3
Move ring 2 from peg 2 to peg 1
Move ring 1 from peg 3 to peg 2
Move ring 1 from peg 2 to peg 1
Move ring 3 from peg 2 to peg 3
Move ring 1 from peg 1 to peg 2
Move ring 1 from peg 2 to peg 3
Move ring 2 from peg 1 to peg 2
Move ring 1 from peg 3 to peg 2
Move ring 1 from peg 2 to peg 1
Move ring 2 from peg 2 to peg 3
Move ring 1 from peg 1 to peg 2
Move ring 1 from peg 2 to peg 3
```

We have a lot more moves for the three-ring ATOH problem than we had for the three-ring TOH problem. The additional constraint on moving rings made the problem significantly harder.

Algorithmic thinking includes analyzing the complexity of algorithms, not just inventing and coding them. Let's compare the number of moves in TOH and ATOH through algorithm analysis.

Divide-and-conquer algorithms have recurrences associated with them. For TOH, we can write:

$$T_n = 2T_{n-1} + 1$$

where T_n is the number of moves required to solve an n-ring TOH and T_{n-1} for an $(n - 1)$-ring TOH. This equation comes directly from the picture and code for TOH shown earlier. $T_0 = 0$ since no rings implies no moves. If we repeatedly apply the equation above, we get $T_1 = 1$, $T_2 = 3$, $T_3 = 7$, $T_4 = 15$, and so on. A little guesswork and checking gives us the answer $T_n = 2^n - 1$.

This is good news if you believe the legend of the Towers of Brahma, because even if the priests could move one disk per second, using the smallest number of moves, it would take them $2^{64} - 1$ seconds or roughly 585 billion years to finish! Our sun won't last that long; in fact, we will need to invent interstellar travel within a few hundred million years, because our sun will likely be much hotter, rendering Earth uninhabitable.

Similarly, for ATOH, using the picture as a guide, we can write:

$$A_n = 3A_{n-1} + 2$$

where A_n is the number of moves required to solve an n-ring ATOH, and A_{n-1} for an $(n - 1)$-ring ATOH. $A_0 = 0$ since no rings implies no moves. If we repeatedly

apply the equation above, we get $A_1 = 2$, $A_2 = 8$, $A_3 = 26$, $A_4 = 80$, and so on. A little guesswork and checking gives us the answer $A_n = 3^n - 1$.

In our examples, for $n = 3$, TOH takes seven moves and ATOH takes twenty-six moves, in keeping with the equations derived.

Relationship to Gray Codes

The reflected binary code (RBC), also known as the Gray code after Frank Gray, is a binary numeral system where two successive values differ in only *one* binary digit, or bit. Gray codes are used extensively to correct errors in digital communication. Interestingly, Gray codes are related to the TOH puzzle.

A Gray code for 1 bit is simply {0, 1}, call it L1. We can construct a Gray code for 2 bits by reversing the 1-bit code to obtain {1, 0}, call it L2. We prefix all entries in L1 with 0, and all entries in L2 with 1. This gives us L1´ = {00, 01} and L2´ = {11, 10}. We concatenate L1´ and L2´ to get a 2-bit Gray code {00, 01, 11, 10}.

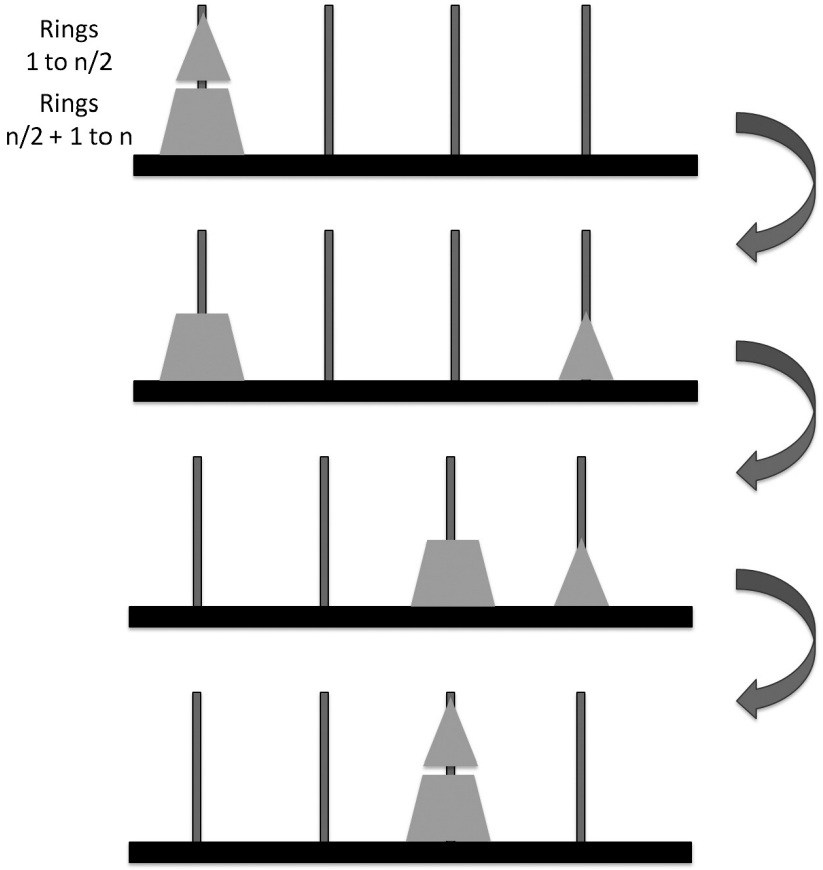
In this way we can generate a 3-bit Gray code {000, 001, 011, 010, 110, 111, 101, 100} from a 2-bit Gray code, then a 4-bit Gray code {0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000}, and so on.

We need an *n*-bit Gray code if we have a TOH problem with *n* rings. And then the Gray code tells us the moves we need to make! There are $2^n - 1$ moves in TOH, corresponding to the $2^n - 1$ transitions in a 2^n-length Gray code. The smallest ring corresponds to the rightmost (least significant) digit and the largest ring to the leftmost (most significant) digit. The ring that moves corresponds to the digit that changes. For example, when we go 000 → 001, we move the smallest ring. But which peg are we moving it to? This matters if there is a choice.

For the smallest ring there are always two pegs it can be moved to. For the other rings there is only one possibility. If the number of rings is odd, the smallest ring cycles along the pegs in the order start → end → middle → start → end → middle, and so on. If the number of rings is even, the order must be: start → middle → end → start → middle → end, and so on. Try it out for three and four rings.

Exercises

Puzzle Exercise 1: Suppose you have a fourth peg. One way to reduce the number of moves required for *n* rings is to split the problem into two *n*/2 ring problems, as shown below. In each of the steps shown, you can invoke the classic Hanoi procedure by choosing three pegs to work with. You can assume *n* is even for this exercise.



For example, in the first step, you invoke classic Hanoi, choosing either of the two pegs in the middle as the single middle peg, with the end peg being the fourth peg.

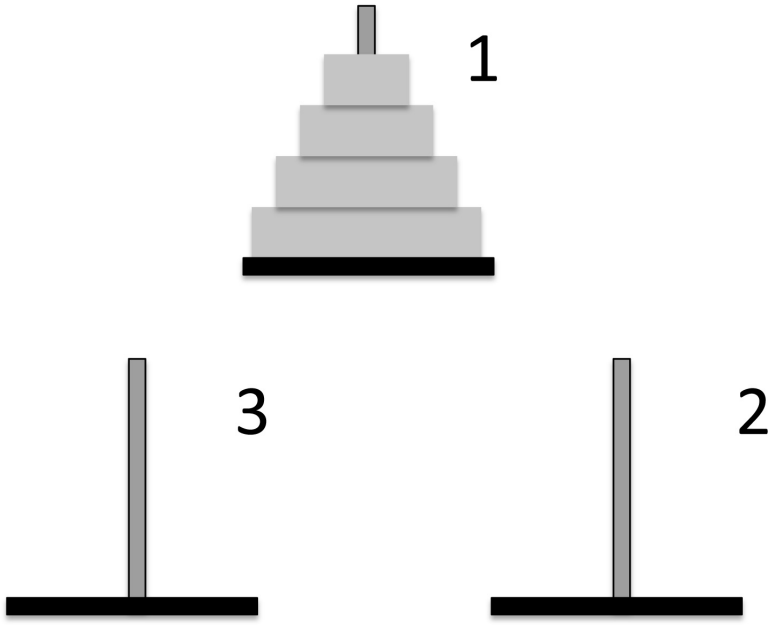
In the second step, you invoke classic Hanoi, except that you have to number the rings appropriately since you are dealing with the rings in the bottom half, numbered from *n*/2 + 1 to *n*, and you have to properly print the moves out.

In the third step, you invoke classic Hanoi, with the start peg being the fourth peg, the destination being the third (the standard end peg in classic Hanoi), and one of the first two pegs as the single middle peg. Code a solution mimicking the steps in the picture. You should require forty-five moves for *n* = 8 in your solution, which is much smaller than the 255 moves required for the three-peg

version. Each of the three classic Hanoi problems that are solved require $2^{8/2} - 1 = 15$ steps.

Puzzle Exercise 2: It turns out you can do better than the solution above. In the first and third steps, you could have used two "middle" pegs. Optimize your code by recursively calling the four-peg pictorial algorithm in the first and third steps. This should reduce the required number of moves to thirty-three for $n = 8$. You can assume that $n = 2^k$ for some k for this exercise. If you are interested in how to reduce the number of moves even further, look for the Frame-Stewart algorithm on Wikipedia. Briefly, the algorithm chooses an optimal $k < n$, and splits the problem into one with k rings and $n - k$ rings.

Puzzle Exercise 3: In Cyclic Hanoi, we are given three pegs numbered 1, 2, and 3, arranged in a circle as shown below, with the clockwise and counterclockwise directions defined as $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $1 \rightarrow 3 \rightarrow 2 \rightarrow 1$, respectively.



Rings may only move clockwise. Given n rings on peg 1, devise and code a recursive procedure to move the rings to peg 2.

You will need two mutually recursive procedures, one that solves the n-ring problem to the neighboring clockwise target peg, and one that solves the n-ring problem to the neighboring counterclockwise target peg.

The clockwise procedure moves $n - 1$ rings counterclockwise $1 \rightarrow 3$, moves

the bottom ring clockwise 1 → 2, and moves $n - 1$ rings counterclockwise 3 → 2. The base case of one ring is simply one clockwise move.

The counterclockwise procedure is more complicated since only clockwise moves are permitted. The base case will correspond to moving a single ring clockwise twice. You will need to invoke both the counterclockwise and clockwise procedures for $n - 1$ rings to make these two moves.

Here's what you should get for $n = 3$.

```
Move ring 1 from peg 1 to peg 2
Move ring 1 from peg 2 to peg 3
Move ring 2 from peg 1 to peg 2
Move ring 1 from peg 3 to peg 1
Move ring 2 from peg 2 to peg 3
Move ring 1 from peg 1 to peg 2
Move ring 1 from peg 2 to peg 3
Move ring 3 from peg 1 to peg 2
Move ring 1 from peg 3 to peg 1
Move ring 1 from peg 1 to peg 2
Move ring 2 from peg 3 to peg 1
Move ring 1 from peg 2 to peg 3
Move ring 2 from peg 1 to peg 2
Move ring 1 from peg 3 to peg 1
Move ring 1 from peg 1 to peg 2
```

13

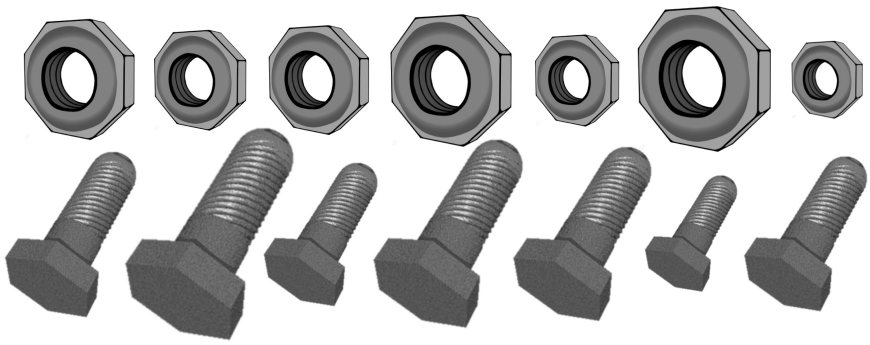
The Disorganized Handyman

A bad handyman always blames his tools.
—Famous proverb

What if my hammer is made of paper? Can I blame it then?
—Author unknown

> Programming constructs and algorithmic paradigms covered in this puzzle: In-place pivoting. Recursive in-place sorting.

A handyman has a whole collection of nuts and bolts of different sizes in a bag. Each nut is unique and has a corresponding unique bolt, but the disorganized handyman has dumped them all into one bag and they are all mixed up (see below). How best to "sort" these nuts and attach each to its corresponding bolt?

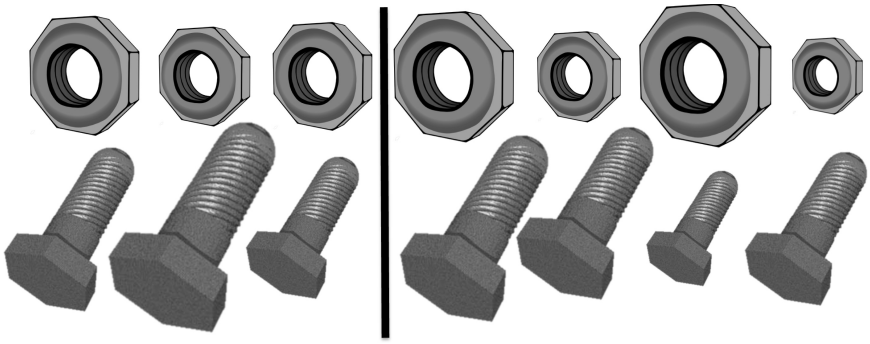


Given *n* nuts and *n* bolts, the handyman can pick an arbitrary nut, try it with

each bolt, and find the one that fits. Then he can put away the nut-bolt pair, and he has a problem of size $n - 1$. This means he has done n comparisons to reduce the problem size by 1. Then $n - 1$ comparisons will shrink the problem size to $n - 2$, and so on. The total number of comparisons required is $n + (n - 1) + (n - 2) + \ldots + 1 = n(n + 1)/2$. You could argue that the last comparison is not required since there will only be one nut and one bolt left, but we will call it a confirmation comparison.

Can we do better in terms of number of comparisons required? More concretely, can we split the nuts and bolts into two sets, each of half the size, so we have two problems of size $n/2$ to work on? This way, if the handyman has a helper, they can work in parallel. Of course, we could apply this strategy recursively to each of the problems of size $n/2$ if there are more kind people willing to help.

Unfortunately, simply splitting the nuts up into two (roughly) equal-sized piles *A* and *B*, and the bolts into similar-sized piles *C* and *D*, does not work. If we group nuts and bolts corresponding to *A* and *C* together into a nut-bolt pile, it is quite possible that a nut in *A* may not fit any bolt in *C*; the correct bolt for that nut is in *D*. Here's an example.



Think of the *A* and *C* piles as the ones to the left, and the *B* and *D* piles as the ones to the right. The biggest bolt is in the left pile (second from left) and the biggest nut is in the right pile (second from right).

Can you think of a recursive divide-and-conquer strategy to solve the nuts-and-bolts problem with significantly fewer than $n(n + 1)/2$ comparisons when n is large?

In devising a divide-and-conquer strategy, we have to determine how to divide the problem so the subproblems are essentially the same as the original problem, but smaller. In the nuts-and-bolts problem, an arbitrary division of nuts (unrelated to the division of bolts) will not work. We have to somehow

guarantee that the subproblems can be solved independently—which means, of course, that each bolt that attaches to each nut in the subproblem's nut-bolt collection has to be in the collection.

Pivoting in Divide-and-Conquer

Pivoting is the key idea that results in a divide-and-conquer algorithm for our problem. We can pick a bolt—call it the *pivot bolt*—and use it to determine which nuts are smaller, which one fits exactly, and which nuts are bigger. We separate the nuts into three piles this way, with the middle pile being of size 1 and containing the paired nut. Therefore, in this process we have discovered one nut-bolt pairing. Using the paired nut—the *pivot nut*—we can now split the bolts into two piles: the bolts that are bigger than the pivot nut, and those that are smaller. The bigger bolts are grouped with the nuts that were bigger than the pivot bolt, and the smaller bolts are grouped with the nuts that were smaller than the pivot bolt.

We now have a pile of "big" nuts and "big" bolts, all together, and a pile of small nuts and small bolts all together. Depending on the choice of the pivot bolt, there will likely be a different number of nuts in the two piles. However, we are guaranteed that the number of nuts is the same as the number of bolts in each pile if the original problem had a matched set of nuts and bolts. Moreover, the nut corresponding to any bolt in the pile is guaranteed to be in the same pile.

In this strategy, we have to make n comparisons with the pivot bolt to split the nuts into two piles. In the process, we discover the pivot nut. We then make $n - 1$ comparisons to split the bolts up and add them to the nut piles. That is a total of $2n - 1$ comparisons. Assuming we chose a pivot nut that was middling in size, we have two problems of size roughly $n/2$. We can subdivide these two problems of size $n/2$, using roughly $2n$ total comparisons, to four problems of size $n/4$.

The cool thing is that the problem sizes halve at each step, rather than only shrinking by 1. For example, suppose $n = 100$. The original strategy requires 5,050 comparisons. In the new strategy, using 199 comparisons, we get two subproblems, each roughly of size 50. Even if we use the original strategy for each of these subproblems, we will only require 1,225 comparisons for each one, for a total of 199 + 1,225 * 2 = 2,649 comparisons. Of course, we can perform recursive divide-and-conquer. In fact, if we can split each problem we encounter roughly in half,[1] the number of comparisons in the recursive strategy will grow as $n \log_2 n$, compared to n^2 in the original strategy. We made a related

observation in the square roots puzzle (puzzle 7).

This puzzle has a deep relationship with perhaps the most widely used sorting algorithm, quicksort. Quicksort relies on the notion of pivoting that we just described.

Relationship to Sorting

Suppose we have a Python list, which is implemented as an array, with unique elements[2] as shown below:

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*

We wish to sort the list in ascending order. We choose an arbitrary pivot element —we'll say *g*, but it could just as easily been the last element, *h*. Now we will partition the list into two sublists, where the left sublist has elements less than *g* and the right sublist has elements greater than *g*. The two sublists are not sorted, that is, the order of elements less than *g* in the left sublist is arbitrary. We can now represent the list as:

Elements less than *g*	*g*	Elements greater than *g*

The beautiful observation is that we can sort the left sublist without affecting the position of *g*, and likewise for the right sublist. Once these two sublists have been sorted, we are done!

Let's look at a possible implementation of the recursive divide-and-conquer quicksort algorithm. We first describe the code for the recursive structure, and then the code for the pivoting step.

```
1.    def quicksort(lst, start, end):
2.        if start < end:
3.            split = pivotPartition(lst, start, end)
4.            quicksort(lst, start, split - 1)
5.            quicksort(lst, split + 1, end)
```

The function `quicksort` takes a Python list corresponding to the array to be sorted, as well as the start and end indices of the list. The list elements go from `lst[start]` to `lst[end]`. We could have assumed that the starting index is 0 and the ending index is the length of the list minus 1, but as you will see, asking for the indices as arguments has the advantage of not requiring a wrapper function like we had to

have in the *N*-queens puzzle (puzzle 10) and the courtyard tiling puzzle (puzzle 11). It is important to note that the procedure modifies the argument list lst to be sorted and does not return anything.

If start equals end, then there is only one element in the list, which does not need sorting—that is our base case and we do not need to modify the list. If there are two or more elements, we have to split the list. The function pivotPartition selects a pivot element in the list (*g* in our example above) and modifies the list so the elements less than the pivot are before the pivot, and the ones greater than the pivot are after the pivot. The index of the pivot element is returned. If we have the index, we can effectively split the list simply by telling the recursive calls what the start and end indices are. This is primarily why we have them as arguments to quicksort. Since the element at the end index is part of the list and we do not have to touch lst[split], the two recursive calls correspond to lst[start] to lst[split - 1] (line 4) and lst[split + 1] to lst[end] (line 5).

All that remains is to implement pivotPartition, which chooses a pivot between the start and end indices and modifies the argument list between the start and end indices appropriately. Here's a first implementation of pivotPartition.

```
1.  def pivotPartition(lst, start, end):
2.      pivot = lst[end]
3.      less, pivotList, more = [], [], []
4.      for e in lst:
5.          if e < pivot:
6.              less.append(e)
7.          elif e > pivot:
8.              more.append(e)
9.          else:
10.             pivotList.append(e)
11.     i = 0
12.     for e in less:
13.         lst[i] = e
14.         i += 1
15.     for e in pivotList:
16.         lst[i] = e
17.         i += 1
18.     for e in more:
19.         lst[i] = e
20.         i += 1
21.     return lst.index(pivot)
```

The first line of the body of the function chooses the pivot element as the last element of the list (line 2). In our nuts-and-bolts problem, we wish to find a middling-size pivot, and here too we would like to choose an element such that about half the other elements are smaller and the other half are bigger. We do not want to search for the best pivot because that may require significant computation. If we assume that the input list is initially randomized, then there is equal probability that any element is a "middle" element. Therefore, we pick the last element as the pivot. Note that the order of the smaller and bigger elements in the split lists will still be random, since we are not sorting them in pivotPartition. We can therefore keep picking the last element as the pivot recursively and we will get split lists that are roughly equal in size. This means that on average, quicksort will only require $n \log_2 n$ comparisons to sort the original list with n elements. In a pathological case it may require n^2 operations—we will explore quicksort's behavior in the exercises.

We have three lists: the elements smaller than the pivot (less), the elements equal to the pivot (pivotList), and the elements bigger than the pivot (more). pivotList is a list because there may be repeated values in the list and one of these may be chosen as a pivot. These three lists are initialized to be empty on line 3. In lines 4–10, scanning the input list lst populates the three lists. In lines 11–20, lst is modified to have elements in less, followed by elements in pivotList, followed by elements in more.

Finally, the index of the pivot element is returned. If there are repeated elements in the list—and, in particular, if the pivot element is repeated—the index of the first occurrence of the pivot is returned. This means that the second recursive call in quicksort (line 5) will operate on a (sub)list whose first elements are equal to the pivot. These elements will remain at the front of the (sub)list since the other elements are all greater than the pivot.

In-Place Partitioning

The above implementation does not exploit the main advantage of quicksort, in that the partitioning step (i.e., going from the original list/array to the one with *g*'s location fixed, and the two sublists unsorted but satisfying ordering relationships with *g*) can be done *without* requiring additional list/array storage.

The merge sort algorithm shown in puzzle 11 only requires $n \log_2 n$ comparisons in the worst case. Merge sort guarantees, during splitting, that two sublists differ in size by at most one element. The merge step of merge sort is

where all the work happens. In quicksort, the pivot-based partitioning or splitting is the workhorse step. The merge step is trivial. Merge sort requires additional storage of a temporary list during its merge step, whereas quicksort does not, as we will show below.

While the selection sort algorithm we coded in puzzle 2 also does not have to make a copy of the list to be sorted, it is quite slow—it has two nested loops, each of which is run approximately n times, where n is the size of the list to be sorted. This means it requires a number of comparisons that grows as n^2, similar to the naive nuts-and-bolts pairing algorithm we discussed that requires $n(n + 1)/2$ comparisons.

Quicksort is one of the most widely deployed sorting algorithms because, on average, it only requires $n \log_2 n$ comparisons, and does not require additional list storage if pivotPartition is implemented cleverly, as shown in the following code.[3]

```
1.  def pivotPartitionClever(lst, start, end):
2.      pivot = lst[end]
3.      bottom = start - 1
4.      top = end
5.      done = False
6.      while not done:
7.          while not done:
8.              bottom += 1
9.              if bottom == top:
10.                 done = True
11.                 break
12.             if lst[bottom] > pivot:
13.                 lst[top] = lst[bottom]
14.                 break
15.         while not done:
16.             top -= 1
17.             if top == bottom:
18.                 done = True
19.                 break
20.             if lst[top] < pivot:
21.                 lst[bottom] = lst[top]
22.                 break
23.     lst[top] = pivot
24.     return top
```

This code is quite different from the first version. The first thing to observe about this code is that it works exclusively on the input list lst and does not allocate additional list/array storage to store list elements, other than the variable pivot, which stores one list element. (The list variables less, pivotList, and more have disappeared.) Furthermore, only list elements between the start and end indices are modified. This procedure uses *in-place* pivoting—the list elements exchange positions, and are not copied from one list to another wholesale as in the first version of the procedure.

It is easiest to understand the procedure with an example. Suppose we want to sort the following list:

```
a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
quicksort(a, 0, len(a) - 1)
```

How exactly is the first pivoting done in-place? The pivot is the last element 1. When pivotPartitionClever is called for the first time, it is called with start = 0 and end = 8. This means that bottom = -1 and top = 8. We enter the outer **while** loop and then the first inner **while** loop (line 7). The variable bottom is incremented to 0. We search rightward from the left of the list for an element that is greater than the pivot element 1. The very first element a[0] = 4 > 1. We copy over this element to a[top], which contains the pivot. At this point, we have element 4 duplicated in the list, but no worries—we know what the pivot is, since we stored it in the variable pivot. If we printed the list and the variables bottom and top after the first inner **while** loop completed, we would see:

```
[4, 65, 2, -31, 0, 99, 83, 782, 4] bottom = 0 top = 8
```

Now we enter the second inner **while** loop (line 15). We search moving leftward from the right of the list at a[7] (the variable top is decremented before the search) for an element that is less than the pivot 1. We keep decrementing top till we see the element 0, at which point top = 4, since a[4] = 0. We copy element 0 to a[bottom = 0]. Remember that a[bottom] was copied over to a[8] before this, so we are not losing any elements in the list. This produces:

```
[0, 65, 2, -31, 0, 99, 83, 782, 4] bottom = 0 top = 4
```

We have taken one element 4 that is greater than the pivot 1 and put it all the way to the right part of the list, and we have taken one element 0 which is less than the pivot 1 and put it all the way to the left part of the list.

We now go into the second iteration of the outer **while** loop. The first inner **while** loop produces:

```
[0, 65, 2, -31, 65, 99, 83, 782, 4] bottom = 1 top = 4
```

From the left we found `65 > 1` and we copied it over to `a[top = 4]`. Next, the second inner **while** loop produces:

```
[0, -31, 2, -31, 65, 99, 83, 782, 4] bottom = 1 top = 3
```

We moved leftward from `top = 4` and discovered `-31 < 1`, and copied it over to `a[bottom = 1]`.

In the second outer **while** loop iteration we moved one element, `65`, to the right part of the list, where all elements to the right of `65` are greater than the pivot `1`. And we moved `-31` to the left part of the list, where all elements to the left of `-31` are less than the pivot `1`.

We begin the third iteration of the outer **while** loop. The first inner **while** loop produces:

```
[0, -31, 2, 2, 65, 99, 83, 782, 4] bottom = 2 top = 3
```

We discovered `a[bottom = 2] = 2 > 1` and moved it to `a[top = 3]`. The second inner **while** loop decrements `top`, sees that it equals `bottom`, and sets `done` to **True**, and we break out of the second inner **while** loop. Since `done` is **True**, we do not continue the outer **while** loop.

We set `a[top = 2] = pivot = 1` (line 23) and return the index of the pivot `1`, which is `2`. The list `a` now looks like:

```
[0, -31, 1, 2, 65, 99, 83, 782, 4]
```

We have indeed pivoted around the element `1`.

Of course, all we have done is split the original list `a` into two lists of sizes 2 and 6. We need to recursively sort these sublists. For the first sublist of 2 elements, we will pick `-31` as the pivot and produce `-31, 0`. For the second sublist, we will pick `4` as the pivot, and the process continues.

Finally, it is important to note that unlike the procedure `pivotPartition`, `pivotPartitionClever` assumes that the pivot is chosen to be the end of the list. So the assignment `pivot = lst[end]` (line 2) is crucial to correctness.

Sort Mania

Sorting is such an important component of data processing that there are hundreds of sorting algorithms. We have not yet mentioned insertion sort and heap sort, both of which are in-place sorting techniques.

Insertion sort needs n^2 comparisons in the worst case, but is relatively efficient

for small lists and lists that are already mostly sorted. Insertion sort always maintains a sorted sublist in the lower positions of the list. Each new element is then inserted back into the previous sublist such that the sorted sublist is one element larger.

Heap sort is a much more efficient version of selection sort and needs *n* log *n* comparisons in the worst case. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, and then continuing with the rest of the list, but it does this efficiently by using a data structure called a heap.

Python provides a built-in sort function for lists. Given a list L, you can simply call L.sort to sort the list. The list L is modified. Internally, L.sort uses an algorithm called *Timsort*, which requires *n* log *n* operations and some temporary list storage. Timsort is not a standalone algorithm but a hybrid, an efficient combination of a number of other algorithms. It splits the list into many lists, which are each sorted using insertion sort and merged together using techniques borrowed from merge sort.

You will likely find that L.sort is significantly faster than the quicksort code in this book, but that's mainly because sort is a built-in function carefully written in a low-level language, not because of algorithmic improvements.

Exercises

Exercise 1: Modify pivotPartitionClever to count the number of element moves and to return the number of moves, in addition to returning the pivot. There are exactly two places where list elements are moved from one location to another in pivotPartitionClever. Add up the moves required in all pivotPartitionClever calls and print the total number after sorting is completed. This means that the procedure quicksort should count the moves made in its pivotPartitionClever call and in the two recursive calls that it makes, and should return this count.

When quicksort runs on our example list a = [4, 65, 2, -31, 0, 99, 83, 782, 1], there are 9 moves total. To further verify your implementation, run quicksort on a list generated by L = list(range(100)), which produces an ascending list of numbers from 0 to 99, and check that no moves are made. Then create a list D sorted in descending order using D = list(range(99, -1, -1)). Run quicksort on the list D.

Give an approximate formula for how many moves quicksort will perform in the case of list D, if D has *n* elements.

Exercise 2: The number of moves is not the best indicator of computational complexity, since moves are only made when the comparisons on lines 12 and 20 in pivotPartitionClever return **True**. Count the number of iterations in both inner **while** loops of pivotPartitionClever using the same methodology as in exercise 1. Verify that there are 24 total iterations for both loops across all recursive calls for list a = [4, 65, 2, -31, 0, 99, 83, 782, 1].

Deterministically generate a "randomly ordered" list of 100 numbers as follows:

```
R = [0] * 100
R[0] = 29
for i in range(100):
    R[i] = (9679 * R[i-1] + 12637 * i) % 2287
```

Determine the number of iterations required for each of the lists, L, D, and R. Give an approximate formula for how many iterations quicksort will need in the case of list D from exercise 1, if D has *n* elements. Explain the difference between the number of iterations required for D versus R.

Hint: Think of what sizes the split lists are in the two cases.

Puzzle Exercise 3: A problem related to sorting is the problem of finding the *k*th smallest element in an unsorted array. We will assume all elements are distinct to avoid the question of what we mean by the *k*th smallest when we have repeated elements. One way to solve this problem is to sort and then output the *k*th element, but we would like something faster.

Notice that in quicksort, after the partitioning step, we can tell which sublist has the item we are looking for, just by looking at their sizes. So we only need to recursively examine one sublist, not two. For instance, say we are looking for the 17th-smallest element in our list. After partitioning, the sublist of elements less than the pivot—call it LESS—has size 100, let's say. Then we just need to find the 17th-smallest element in LESS. If the LESS sublist has size exactly 16 then we just return the pivot. On the other hand, if the LESS sublist has size 10, then we need to look for the 17th-smallest element of the original list in GREATER, which is the sublist containing elements greater than the pivot.

Modify quicksort to code quickselect as described above.

Hint: You will not need to modify *k* in either recursive call, nor do you need to modify pivotPartitionClever.

Notes

1. This is not trivial, since we have to pick the nut that looks like roughly half the nuts would be larger and half would be smaller in each pile that we want to split.

2. The sorting algorithm and the code that we will present will work for non-unique elements, but it is easier to describe the algorithm assuming unique elements.

3. Other recursive sorting algorithms that only require $n \log n$ comparisons typically need to allocate additional memory that grows with the size of the list to be sorted.

14

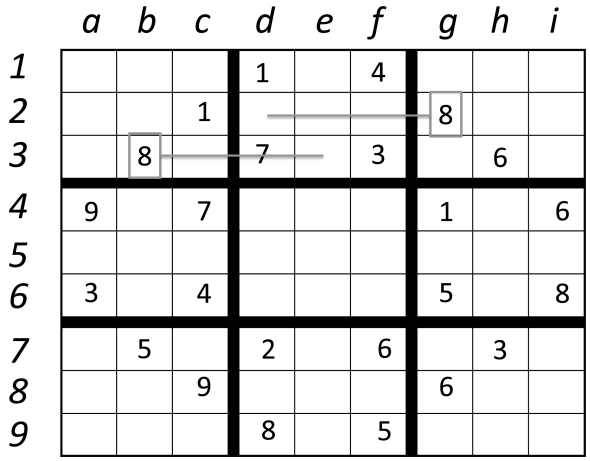
You Won’t Want to Play Sudoku Again

Thanks to modern computers, brawn beats brain.
—Srini Devadas

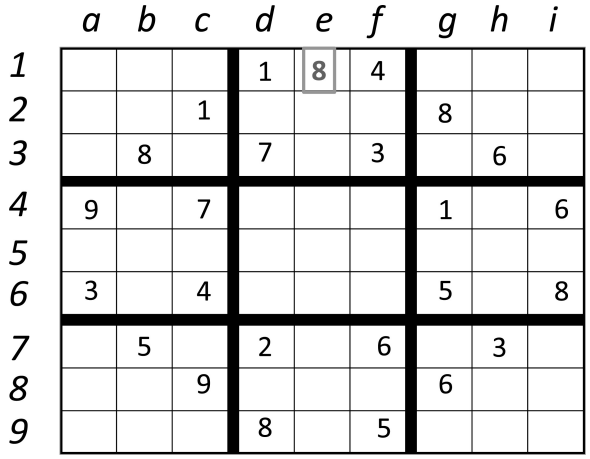
> Programming constructs and algorithmic paradigms covered in this puzzle: Global variables. Sets and set operations. Exhaustive recursive search with implications.

Sudoku is a popular puzzle that involves number placement. You are given a partially filled 9 × 9 grid of digits from 1 to 9. Your goal is to fill out the entire grid with digits from 1 to 9 respecting the following rules: each column, each row, and each of the nine 3 × 3 subgrids or sectors that compose the grid can contain only one occurrence of each of the digits from 1 to 9.

These constraints are used to determine the missing numbers. In the puzzle below, several subgrids have missing numbers. Scanning rows (or columns) can tell us where to place a missing number in a sector.



In the example above, we can determine the position of the 8 in the top middle sector. The 8 cannot be placed in the middle or bottom rows of the middle sector and can only be placed as shown in the diagram below.



Our goal is to write a sudoku solver that can do a recursive search for numbers to be placed in the missing positions. The basic solver does not follow a human strategy such as the one above. It guesses a number at a particular location and determines whether the guess violates a constraint. If not, it proceeds to guess other numbers at other positions. If it detects a violation, it changes the most recent guess. This is similar to the *N*-queens search in puzzle 10.

We want a recursive sudoku solver that solves any sudoku puzzle regardless of how many numbers are filled in. Then we will add “human intelligence” to the solver.

Can you code a recursive sudoku solver following the structure of the N-queens code from puzzle 10?

Recursive Sudoku Solving

The following code is the top-level routine for a basic recursive sudoku solver. The grid is represented by a two-dimensional array/list called grid, and a value of 0 means the location is empty. Grid locations are filled in through a process of a systematic ordered search for empty locations, guessing values for each location, and backtracking (undoing guesses) if they are incorrect.

```
1.  backtracks = 0

2.  def solveSudoku(grid, i=0, j=0):
3.      global backtracks
4.      i, j = findNextCellToFill(grid)
5.      if i == -1:
6.          return True
7.      for e in range(1, 10):
8.          if isValid(grid, i, j, e):
9.              grid[i][j] = e
10.             if solveSudoku(grid, i, j):
11.                 return True
12.             backtracks += 1
13.             grid[i][j] = 0
14.     return False
```

solveSudoku takes three arguments, and for convenience of invocation, we have provided default parameters of 0 for the last two arguments. This way, for the initial call we can simply call solveSudoku(input) on an input grid input. The last two arguments will be set to 0 for this call, but will vary for the recursive calls depending on the empty squares in input. (This is similar to what we did with the quicksort procedure in puzzle 13.)

Procedure findNextCellToFill, which will be shown and explained later, finds the first empty square (value 0) by searching the grid in a predetermined order. If the procedure cannot find an empty square, the puzzle is solved.

Procedure isValid, which will also be shown and explained later, checks that the current, partially filled grid does not violate the rules of sudoku. This is reminiscent of noConflicts (in puzzles 4 and 10), which also worked with partial configurations, that is, configurations with fewer than *N* queens.

The first important point about solveSudoku is that there is only one copy of grid that is being operated on and modified. solveSudoku is therefore an *in-place* recursive search exactly like *N*-queens (puzzle 10). Because of this, we have to

change the value of the position that was filled in with an incorrect number (line 9) back to 0 (line 13), after the recursive call for a particular guess returns **False** and the loop continues. Observe that we only really need to do this after the **for** loop terminates since the next iteration of the **for** loop overwrites grid[i][j] with a new guess. If an invocation of solveSudoku fails because an earlier guess was incorrect and all recursive calls fail, we need to make sure grid has not been changed by the invocation before returning **False**. The following code will work just as well, where line 13 is outside the **for** loop.

```
 7.    for e in range(1, 10):
 8.        if isValid(grid, i, j, e):
 9.            grid[i][j] = e
10.            if solveSudoku(grid, i, j):
11.                return True
12.                backtracks += 1
13.    grid[i][j] = 0
14.    return False
```

One programming construct that you might not have seen before is global. Global variables retain state across function invocations and are convenient to use when we want to keep track of how many recursive calls are made, for instance. We use backtracks as a global variable, initially setting it to zero (at the top of the file), and incrementing it each time we realize we have made an incorrect guess that we need to undo. Note that, to use backtracks in sudokuSolve, we have to declare it global within the function.

Computing the number of backtracks is a great way of measuring performance independent of the computing platform. The more backtracks, the longer the program typically takes to run.

Now, let's look at the procedures invoked by sudokuSolve. findNextCellToFill follows a prescribed order in searching for an empty location, going column by column, starting with the leftmost column and moving rightward. Any order can be used, as long as we ensure that we will not miss any empty values in the current grid at any point in the recursive search.

```
1.    def findNextCellToFill(grid):
2.        for x in range(0, 9):
3.            for y in range(0, 9):
4.                if grid[x][y] == 0:
5.                    return x, y
6.        return -1, -1
```

The procedure returns the grid location of the first empty location, which could be 0, 0 all the way to 8, 8. Therefore, we return -1, -1 if there are no empty locations.

The procedure isValid that follows embodies the rules of sudoku. It takes a partially filled sudoku grid and a new entry e at grid[i, j], and checks whether filling in this entry violates any rules.

```
1.  def isValid(grid, i, j, e):
2.      rowOk = all([e != grid[i][x] for x in range(9)])
3.      if rowOk:
4.          columnOk = all([e != grid[x][j] for x in range(9)])
5.          if columnOk:
6.              secTopX, secTopY = 3 *(i//3), 3 *(j//3)
7.              for x in range(secTopX, secTopX+3):
8.                  for y in range(secTopY, secTopY+3):
9.                      if grid[x][y] == e:
10.                         return False
11.             return True
12.     return False
```

The procedure first checks that each row does not already have an element numbered e (line 2). It does this by using the **all** operator. Line 2 is equivalent to iterating through grid[i][x] for x from 0 through 8 and returning **False** if any entry is equal to e, and returning **True** otherwise. If this check passes, the column corresponding to j is checked on line 4. If the column check passes, we determine the sector that grid[i, j] corresponds to (line 6). We then check if any of the existing numbers in the sector are equal to e (lines 7–10).

Note that isValid, like noConflicts, only checks whether a new entry violates sudoku rules, since it focuses on the row, column, and sector of the new entry. If for example i = 2, j = 2, and e = 2, it does not check that the ith row does not already have two 3's on it, for instance. It is therefore important to call isValid each time an entry is made, and solveSudoku does that.

Finally, here is a simple printing procedure so we can output something that (sort of) looks like a solved sudoku puzzle.

```
1.  def printSudoku(grid):
2.      numrow = 0
3.      for row in grid:
4.          if numrow % 3 == 0 and numrow != 0:
```

```
5.                  print (' ')
6.              print (row[0:3], ' ', row[3:6], ' ', row[6:9])
7.              numrow += 1
```

Line 5 prints a space to create a line spacing after three rows are printed. Remember that each **print** statement produces output on a different line if we do not set end = ''.[1]

We are now ready to run the sudoku solver. Here's an input puzzle given as a two-dimensional array/list:

```
input = [[5, 1, 7, 6, 0, 0, 0, 3, 4],
         [2, 8, 9, 0, 0, 4, 0, 0, 0],
         [3, 4, 6, 2, 0, 5, 0, 9, 0],
         [6, 0, 2, 0, 0, 0, 0, 1, 0],
         [0, 3, 8, 0, 0, 6, 0, 4, 7],
         [0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 9, 0, 0, 0, 0, 0, 7, 8],
         [7, 0, 3, 4, 0, 0, 5, 6, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

We run:

```
solveSudoku(input)
printSudoku(input)
```

This produces:

```
[5, 1, 7]   [6, 9, 8]   [2, 3, 4]
[2, 8, 9]   [1, 3, 4]   [7, 5, 6]
[3, 4, 6]   [2, 7, 5]   [8, 9, 1]

[6, 7, 2]   [8, 4, 9]   [3, 1, 5]
[1, 3, 8]   [5, 2, 6]   [9, 4, 7]
[9, 5, 4]   [7, 1, 3]   [6, 8, 2]

[4, 9, 5]   [3, 6, 2]   [1, 7, 8]
[7, 2, 3]   [4, 8, 1]   [5, 6, 9]
[8, 6, 1]   [9, 5, 7]   [4, 2, 3]
```

Check to make sure the puzzle was solved correctly. On the puzzle input, sudokuSolve takes 579 backtracks. If we run sudokuSolve on a different puzzle, shown below, it takes 6,363 backtracks. The second puzzle is the same as the first puzzle but with a few numbers removed, shown with **0** rather than 0. This makes the puzzle harder for our solver.

```
inp2 = [[5, 1, 7, 6, 0, 0, 0, 3, 4],
        [0, 8, 9, 0, 0, 4, 0, 0, 0],
```

[3, **0**, 6, 2, 0, 5, 0, 9, 0],
[6, 0, **0**, 0, 0, 0, 0, 1, 0],
[0, 3, **0**, 0, 0, 6, 0, 4, 7],
[0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 9, 0, 0, 0, 0, 0, 7, 8],
[7, 0, 3, 4, 0, 0, 5, 6, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0]]

The basic solver does not infer the position for the 8 as we did in our very first sudoku example. The same technique can be expanded by using information from perpendicular rows and columns. Let's see where we can place a 1 in the top right box in the example below. Row 1 and row 2 contain 1's, which leaves two empty squares at the bottom of our focus box. However, square *g*4 also contains 1, so no additional 1 is allowed in column *g*.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
1				1		4			
2			1				8		
3		8		7		3		6	
4	9		7				1		6
5									
6	3		4				5		8
7		5		2		6		3	
8			9				6		
9				8		5			

This means that square *i*3 is the only place left for 1 (see below).

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
1				1		4			
2			1				8		
3		8		7		3		6	1
4	9		7				1		6
5									
6	3		4				5		8
7		5		2		6		3	
8			9				6		
9				8		5			

Can you augment the recursive sudoku solver to make these or other kinds of inferences?

Implications during Recursive Search

Because the current state of the grid *implies* a position for the 1 in the above example, we will say that the solver has inferred an implication if it can make such an observation. We will show how to augment our solver to infer implications—though not in the exact way we described in our examples—and see how much more efficient the solver becomes. We can do this by measuring the number of backtracks with and without implications. Implications more quickly determine whether a particular assignment of values to empty squares is correct.

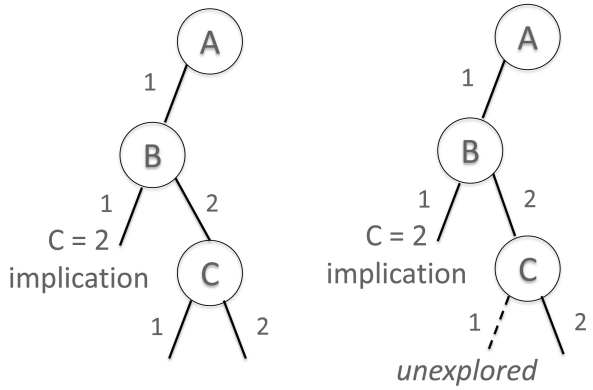
Several changes must be made to the solver to correctly implement this inference. One or more implications can be inferred once a particular grid location is assigned a value. The recursive search code in the optimized solver needs to be slightly different to accommodate implications.

```
1.  backtracks = 0

2.  def solveSudokuOpt(grid, i=0, j=0):
3.      global backtracks
4.      i, j = findNextCellToFill(grid)
5.      if i == -1:
6.          return True
7.      for e in range(1, 10):
8.          if isValid(grid, i, j, e):
9.              impl = makeImplications(grid, i, j, e)
10.             if solveSudoku(grid, i, j):
11.                 return True
12.             backtracks += 1
13.             undoImplications(grid, impl)
14.     return False
```

The only changes are on lines 9 and 13. On line 9, not only are we filling in the `grid[i][j]` entry with `e`, we are also making implications and filling in other grid locations. All of these have to be "remembered" in the implication list `impl`. On line 13, we have to undo *all* the changes made to the grid because the `grid[i][j] = e` guess was incorrect. This line *has* to be inside the **for** loop because implications across iterations of the **for** loop can correspond to different grid locations.

Storing the assignment and implications inferred, so we can roll them all back if the assignment does not work, is important for correctness—otherwise we might not explore the entire search space, and therefore not find a solution. To understand this, look at the following figure.



Think of *A*, *B*, and *C* above as being grid locations, and assume we only have two numbers 1 and 2 that are possible entries. (We have a simplified situation for illustration purposes.) Suppose we assign *A* = 1, *B* = 1, and then *C* = 2 is implied. After exploring the *A* = 1, *B* = 1 branch fully, we backtrack to *A* = 1, *B* = 2. Here, we need to explore *C* = 1 and *C* = 2, as in the picture to the left, not just *C* = 2, as shown on the right. What might happen is that *C* is still set to 2 in the *B* = 2 branch, and we—in effect—only explore the *B* = 2, *C* = 2 branch. So we need to roll back all the implications associated with an assignment.

The procedure undoImplications is short, as shown below.

```
1.    def undoImplications(grid, impl):
2.        for i in range(len(impl)):
3.            grid[impl[i][0]][impl[i][1]] = 0
```

impl is a list of 3-tuples, where each 3-tuple has the form (i, j, e), meaning that grid[i][j] = e. In undoImplications we don't care about the third item, e, since we want to empty out the entry.

makeImplications is more involved since it performs significant analysis. The pseudocode for makeImplications is below. The line numbers are for the Python code that is shown after the pseudocode.

For each sector (subgrid):

Find missing elements in the sector (lines 8–12).

Attach set of missing elements to each empty square in sector (lines 13–16).

For each empty square *S* in sector: (lines 17–18)

Subtract all elements on *S*'s row from missing elements set (lines 19–22).

Subtract all elements on *S*'s column from missing elements set (lines 23–26).

If missing elements set is a *single* value, then: (line 27)

Missing square value can be implied to be that value (lines 28–31).

```
1.  sectors = [[0, 3, 0, 3], [3, 6, 0, 3], [6, 9, 0, 3],
                [0, 3, 3, 6], [3, 6, 3, 6], [6, 9, 3, 6],
                [0, 3, 6, 9], [3, 6, 6, 9], [6, 9, 6, 9]]

2.  def makeImplications(grid, i, j, e):
3.      global sectors
4.      grid[i][j] = e
5.      impl = [(i, j, e)
6.      for k in range(len(sectors)):
7.          sectinfo = []
8.          vset = {1, 2, 3, 4, 5, 6, 7, 8, 9}
9.          for x in range(sectors[k][0], sectors[k][1]):
10.             for y in range(sectors[k][2], sectors[k][3]):
11.                 if grid[x][y] != 0:
12.                     vset.remove(grid[x][y])
13.         for x in range(sectors[k][0], sectors[k][1]):
14.             for y in range(sectors[k][2], sectors[k][3]):
15.                 if grid[x][y] == 0:
16.                     sectinfo.append([x, y, vset.copy()])
17.         for m in range(len(sectinfo)):
18.             sin = sectinfo[m]
19.             rowv = set()
20.             for y in range(9):
21.                 rowv.add(grid[sin[0]][y])
22.             left = sin[2].difference(rowv)
23.             colv = set()
24.             for x in range(9):
25.                 colv.add(grid[x][sin[1]])
26.             left = left.difference(colv)
27.             if len(left) == 1:
28.                 val = left.pop()
29.                 if isValid(grid, sin[0], sin[1], val):
30.                     grid[sin[0]][sin[1]] = val
31.                     impl.append((sin[0], sin[1], val))
```

```
32.             return impl
```

Line 1 declares variables that give the grid indices of each of the nine sectors. For example, the middle sector 4 varies from 3 to 5 inclusive in the *x* and *y* coordinates. This is helpful in ranging over the grid while staying within a sector.

This code uses the set data structure in Python. An empty set is declared using set(), as opposed to an empty list, which is declared as []. A set cannot have repeated elements. Note that even if we included a number, say 1, twice in the declaration of a set, it would only be included once in the set. V = {1, 1, 2} is the same as V = {1, 2}.

Line 8 declares a set vset that contains numbers 1 through 9. In lines 8–12, we go through the elements in the sector and remove these elements from vset using the remove function. We wish to append this missing element set to *each* empty square, so we create a list sectinfo of 3-tuples. Each 3-tuple has the *x*, *y* coordinates of the empty square in the sector, and a copy of the set of missing elements in the sector. We need to make copies of sets because these copies will diverge in their membership later in the algorithm.

For each empty square in the sector, we look at the corresponding 3-tuple in sectinfo (line 18). The elements in the corresponding row are removed from the missing element set given by sin[2], the third element of the 3-tuple, by using the set difference function (line 22). Similarly for the column associated with the empty square. The remaining elements are stored in the set left.

If the set left has cardinality 1 (line 27), we may have an implication. Why are we not guaranteed an implication? The way we have written the code, we compute the missing elements for each sector, and try to find implications for each empty square in the sector. The first implication will hold, but once we make one particular implication, the sector changes, as does the missing elements set. So further implications computed using stale missing elements information may not be valid. This is why we check if the implication violates the rules of sudoku (line 29) before including it in the implication list impl.

This optimization shrinks the number of backtracks from 579 to 10 for the puzzle input and from 6,363 to 33 for puzzle inp2. Of course, from an execution standpoint, both versions run in fractions of a second. This is one reason we included the functionality of counting backtracks in the code: so you could see that the optimizations do help reduce the guessing required.

Difficulty of Sudoku Puzzles

A Finnish mathematician, Arto Inkala, claimed in 2006 that he had created the world's hardest sudoku puzzle, and followed it up in 2010 with a claim of an even harder puzzle. The first puzzle takes the unoptimized solver 335,578 backtracks, and the second takes 9,949 backtracks! The solver finds solutions in a matter of seconds. To be fair, Inkala was predicting human difficulty. The following is Inkala's 2010 puzzle.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
1			5	3					
2	8							2	
3		7			1		5		
4	4					5	3		
5		1			7				6
6			3	2				8	
7		6		5					9
8			4					3	
9						9	7		

Peter Norvig has written sudoku solvers that use constraint programming techniques significantly more sophisticated than the simple implications we have presented here. As a result, the amount of backtracking required even for difficult puzzles is quite small.

We suggest you find sudoku puzzles with different levels of difficulty, from easy to very hard, and explore how the number of backtracks required in the basic solver and the optimized solver change as the difficulty increases. You might be surprised by what you find!

Exercises

Exercise 1: We'll improve our optimized (classic) sudoku solver in this exercise. Each time we discover an implication, the grid changes, and we may find other implications. In fact, this is the way humans solve sudoku puzzles. Our optimized solver goes through all the sectors, trying to find implications, and then stops. If we find an implication in one "pass" through the grid sectors, we can try repeating the entire process (lines 6–31) until we can't find implications (i.e., can't add to our data structure impl). Code this improved sudoku solver.

You should get backtracks = 2 in your improved solver for sudoku puzzle inp2, down from 33.

Puzzle Exercise 2: Modify the basic sudoku solver to work with diagonal sudoku, where there is an additional constraint that all the numbers 1 through 9 must appear on both diagonals.

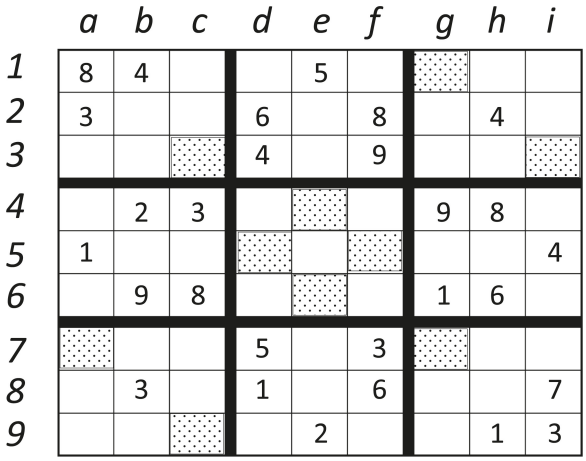
The following is a diagonal sudoku puzzle:

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
1	1		5	7		2	6	3	8
2	2					6			5
3		6	3	8	4		2	1	
4		5	9	2		1	3	8	
5			2		5	8			9
6	7	1			3		5		2
7			4	5	6		7	2	
8	5					4		6	3
9	3	2	6	1		7			4

And here is its solution:

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
1	1	4	5	7	9	2	6	3	8
2	2	8	7	3	1	6	4	9	5
3	9	6	3	8	4	5	2	1	7
4	4	5	9	2	7	1	3	8	6
5	6	3	2	4	5	8	1	7	9
6	7	1	8	6	3	9	5	4	2
7	8	9	4	5	6	3	7	2	1
8	5	7	1	9	2	4	8	6	3
9	3	2	6	1	8	7	9	5	4

Puzzle Exercise 3: Modify the basic sudoku solver to work with even sudoku, which is similar to classic sudoku, except that particular squares have to have even numbers. An example is below.

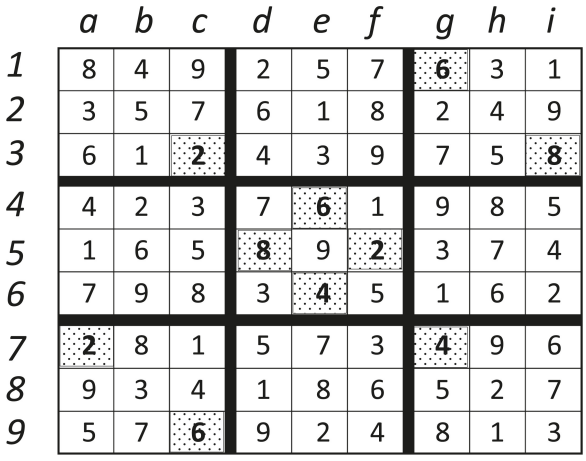


	a	b	c	d	e	f	g	h	i
1	8	4			5				
2	3			6		8		4	
3				4		9			
4		2	3				9	8	
5	1								4
6		9	8				1	6	
7				5		3			
8		3		1		6			7
9					2			1	3

The grayed-out blank squares have to contain even numbers; the other squares can contain either odd or even numbers. To represent the puzzle using a two-dimensional list, we will use 0's as before to indicate blank squares without additional constraints, and −2's to indicate that the square is blank and has to contain an even number. The input list for the above puzzle is:

```
even = [[8, 4, 0, 0, 5, 0,-2, 0, 0],
        [3, 0, 0, 6, 0, 8, 0, 4, 0],
        [0, 0,-2, 4, 0, 9, 0, 0,-2],
        [0, 2, 3, 0,-2, 0, 9, 8, 0],
        [1, 0, 0,-2, 0,-2, 0, 0, 4],
        [0, 9, 8, 0,-2, 0, 1, 6, 0],
        [-2,0, 0, 5, 0, 3,-2, 0, 0],
        [0, 3, 0, 1, 0, 6, 0, 0, 7],
        [0, 0,-2, 0, 2, 0, 0, 1, 3]]
```

The solution to the above puzzle is:



	a	b	c	d	e	f	g	h	i
1	8	4	9	2	5	7	**6**	3	1
2	3	5	7	6	1	8	2	4	9
3	6	1	**2**	4	3	9	7	5	**8**
4	4	2	3	7	**6**	1	9	8	5
5	1	6	5	**8**	9	**2**	3	7	4
6	7	9	8	3	**4**	5	1	6	2
7	**2**	8	1	5	7	3	**4**	9	6
8	9	3	4	1	8	6	5	2	7
9	5	7	**6**	9	2	4	8	1	3

Note

1. Python 2.x treats `end = ''` differently and doesn’t produce the same output, but it does produce a readable output corresponding to the solved sudoku puzzle.

15

Counting the Ways You Can Count Change

Money often costs too much.
—Ralph Waldo Emerson

> Programming constructs and algorithmic paradigms covered in this puzzle: Recursive generation of combinations.

You have piles of money. In fact, you have piles of almost every kind of dollar bill that was ever printed. This includes $1, $2, $5, $10, $20, $50, and $100 bills.[1]

You owe your friend $6. Your friend knows you have piles of cash, and asks if you know how many different ways you can pay using different denominations of bills. You think for a while, collecting different bills, checking what they add up to, and figure out that there are only five different ways:

$1, $1, $1, $1, $1, $1
$1, $1, $1, $1, $2
$1, $1, $2, $2
$1, $5
$2, $2, $2

(Although each bill has a different serial number, you view two bills as identical if they have the same denomination.)

All of a sudden you realize you actually owe your friend $16, remembering the $20 dinner you had together recently, after which you couldn't pay your share because you had forgotten your wallet.

What are all the different ways you can pay your friend $16? How many ways are there?

Recursive Bill Selection

We will explore different ways of selecting bills, keeping track of the total value of the bills selected. As long as the total value is less than our target value, we will add more bills. If we exceed the target value, we simply discard the solution. If we get the exact target total, we print (or store) the solution and keep going. Remember, we want all possible solutions that exactly reach the target value.

The easiest way to implement such an exploration is to use recursion—which probably does not surprise you by now. Here's code that recursively enumerates possible solutions.

```
1.  def makeChange(bills, target, sol = []):
2.      if sum(sol) == target:
3.          print (sol)
4.          return
5.      if sum(sol) > target:
6.          return
7.      for bill in bills:
8.          newSol = sol[:]
9.          newSol.append(bill)
10.         makeChange(bills, target, newSol)
11.     return
```

The first argument to the function is a list of bill denominations in `bills`, for example, `[1, 2, 5]`. The second argument is the target value, and the third is the solution `sol`, which is a list of chosen bill denominations. Lines 2–4 correspond to a base case where we discover a solution and print it. In this particular implementation, we are not storing or counting possible solutions, merely printing them out as we discover them.

Lines 5–6 are another base case, where we have exceeded the target value. We simply discard this non-solution. There is no reason to pay your friend more than you owe.

Lines 7–10 explore the solution space. For each denomination listed in bills, we copy the current solution to a list newSol, and add a bill of that denomination to it. Then we recursively call makeChange with the longer list newSol.

Since we are iterating over different bill denominations, we need to make a copy of sol in newSol (line 8). Why? Suppose we did not make a copy of sol in newSol, and just used sol everywhere in the code. Let's say bills = [1, 2, 5] and we are exploring solutions with sol = [1]. We add 1 to sol to get sol = [1, 1]. We go ahead and make recursive calls with this sol, and each call adds to the list sol. Once these calls return—some solutions may have been found, and non-solutions discarded—we go to the next iteration of the loop. We would like for sol to be [1], so we can add 2 to it and explore solutions with sol = [1, 2]. Unfortunately, sol could be a long list (depending on the target value) since we have not removed any elements from it. Adding another bill to this list will likely mean exceeding the target value. Copying sol to newSol ensures that we properly explore the entire space.

Suppose we run:

```
bills = [1, 2, 5]
makeChange(bills, 6)
```

This outputs:

```
[1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 2]
[1, 1, 1, 2, 1]
[1, 1, 2, 1, 1]
[1, 1, 2, 2]
[1, 2, 1, 1, 1]
[1, 2, 1, 2]
[1, 2, 2, 1]
[1, 5]
[2, 1, 1, 1, 1]
[2, 1, 1, 2]
[2, 1, 2, 1]
[2, 2, 1, 1]
[2, 2, 2]
[5, 1]
```

There is a problem here. The program is repeating solutions and produces 15 lines of output. It seems to think that the order of the bills matters—that four $1 bills, then a $2 bill, is different from three $1 bills, then a $2 bill, then a $1 bill. It does realize that bills of the same denomination are identical, and does not generate 6! = 720 different [1, 1, 1, 1, 1, 1] solutions, thank goodness!

Eliminating Repetition

How can we eliminate this repetition of solutions? We can use the natural ordering of the bill denominations to generate solutions of a particular form. We won't generate any solution of the form [*a*, *b*] or [*a*, *b*, *c*] where $b < a$ or $c < b$ or $c < a$. Each solution will correspond to a possibly different but always *nondecreasing* order of denomination. Certainly, we need to allow solutions [*a*, *b*, *c*], where $b \geq a$, or $c \geq b$, or $c \geq b \geq a$. But we will preclude a solution like [1, 1, 1, 2, 1] because it is not a nondecreasing sequence, and similarly, [5, 1]. The allowed solutions for these two cases will be [1, 1, 1, 1, 2] and [1, 5].

We therefore make a small, subtle modification to makeChange. We add an argument that corresponds to the highest denomination that has been generated thus far. In recursive exploration, we will only add denominations that are equal to or greater than the highest denomination added thus far, and not ones that are lower. On our initial call to the modified procedure, we will set this new argument to the lowest-denomination bill, which in our examples is $1. This means we can keep adding $1 bills, but if we ever add a $2 bill, we cannot go back to adding a $1 bill.

This algorithmic change is embodied in the function makeSmartChange below.

```
1..   def makeSmartChange(bills, target, highest, sol = []):
2.        if sum(sol) == target:
3.            print (sol)
4.            return
5.        if sum(sol) > target:
6.              return
7.        for bill in bills:
8.            if bill >= highest:
9.                newSol = sol[:]
10.               newSol.append(bill)
11.               makeSmartChange(bills, target, bill, newSol)
12.       return
```

Note the additional highest argument that we discussed (line 1). Exactly one line has been added to makeChange—line 8. When we generate a new solution, we make sure that the bill that is appended has a value greater than or equal to highest. We also have to change the recursive call on line 11 since we have added an argument to the function. The argument has to be bill—we can't leave it as highest, because bill may be greater than highest as we go through the **for** loop (lines

7–11). But once we have added `bill >= highest` and made a recursive call, inside that recursive call, `highest` will equal `bill`.

This modification fixes the problem with `makeChange`. If we run:

```
bills = [1, 2, 5]
makeSmartChange(bills, 6, 1)
```

The program produces:

```
[1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 2]
[1, 1, 2, 2]
[1, 5]
[2, 2, 2]
```

This set of solutions you knew already. But if we run:

```
bills2 = [1, 2, 5, 10]
makeSmartChange(bills2, 16, 1)
```

We get twenty-five different ways of counting $16, namely:

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 5]
[1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 2, 2, 5]
[1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2]
[1, 1, 1, 1, 1, 1, 5, 5]
[1, 1, 1, 1, 1, 1, 10]
[1, 1, 1, 1, 1, 2, 2, 2, 5]
[1, 1, 1, 1, 2, 2, 2, 2, 2, 2]
[1, 1, 1, 1, 2, 5, 5]
[1, 1, 1, 1, 2, 10]
[1, 1, 1, 2, 2, 2, 2, 5]
[1, 1, 2, 2, 2, 2, 2, 2, 2]
[1, 1, 2, 2, 5, 5]
[1, 1, 2, 2, 10]
[1, 2, 2, 2, 2, 2, 5]
[1, 5, 5, 5]
[1, 5, 10]
[2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 5, 5]
[2, 2, 2, 10]
```

Nice! Be careful when running this code—the number of solutions explodes as the target value increases, especially if you have low-denomination bills.

Making Change with the Fewest Bills

Suppose you are interested in using the fewest possible bills to pay your friend. The output produced by our code clearly has that information in it. In the $16 example, for instance, you pay your friend using three bills: $1, $5, and $10.

Your first impulse, understandably, may be to try a greedy algorithm. The greedy algorithm would pick the highest-denomination bill that is less than the money owed, lower the target appropriately, and repeat the process. It certainly works for the $16 case: $10 is picked first, then $5, then $1. What if you lived in Simbuktu, where there is an $8 bill? The optimal solution would be two $8 bills, and the greedy algorithm will miss that. Of course, makeSmartChange will produce the two $8 bills solution as one of the valid ones.

One way of always finding the solution with the fewest bills is to run makeSmartChange and pick the minimum cardinality solution during the enumerative process. Exercise 3 asks you do this.

Exercises

Exercise 1: Modify makeSmartChange so that rather than printing each solution, it merely counts the number of distinct solutions and returns the count. You can use a global variable count that counts solutions across recursive calls. There are more efficient ways of simply counting the number of possible solutions[2]—your job here is to modify the given code.

Puzzle Exercise 2: Suppose you are not as rich as we made you out to be, and you only have a limited number of bills of different denominations. For example, the representation of your stash is:

```
yourMoney = [(1, 11), (2, 7), (5, 9), (10, 10), (20, 4)]
```

where the first entry in each tuple in the list is the bill denomination, and the second entry is the number of bills you have of that denomination. In the above example, you have eleven $1 bills and four $20 bills, for instance. Modify the program so it only outputs solutions that you are capable of constructing with your somewhat limited amount of money.

An easy method is to generate all the solutions as before, and discard—without printing—the ones that violate the numeric constraints. We encourage you to follow the more elegant and efficient method of discarding illegal partial

solutions *during* the recursive search. Illegal partial solutions are less than the target value, but violate one or more numeric constraints.

If you run:

```
money = [(1, 3), (2, 3), (5, 1)]
makeSmartChange(money, 6, 1)
```

It should produce the three solutions below:

```
[1, 1, 2, 2]
[1, 5]
[2, 2, 2]
```

While there are three $1 bills, we cannot produce $6 (the target) using all three of them, so the solutions use either two or one $1 bills.

Exercise 3: Since a greedy algorithm doesn't produce the fewest number of bills that meet the target value, modify your code from exercise 2 to return *one* solution with the fewest bills of any denomination used. Again, do not store or print all possible solutions. Rather, just store the current best solution found, corresponding to the fewest bills, and update it during recursive execution as appropriate. Note that you will have a nontrivial best solution only after finding a first solution that meets the target value.

We'll explore a way of making the above strategy significantly more efficient in exercise 4 of puzzle 18.

Notes

1. Sadly, you don't have $3 bills, which haven't been printed since about 1800, but the rest of your collection keeps you happy.

2. Arguably, the title of puzzle 15 should have been "Enumerating the Ways You Can Count Change." See exercise 5, puzzle 18 for the real counting puzzle.

16

Greed Is Good

Greed is right, greed works.
—Gordon Gekko, *Wall Street*, 1987

> Programming constructs and algorithmic paradigms covered in this puzzle: Functions as arguments. Greedy algorithms.

A greedy algorithm is an algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage, with the hope of finding a global optimum.

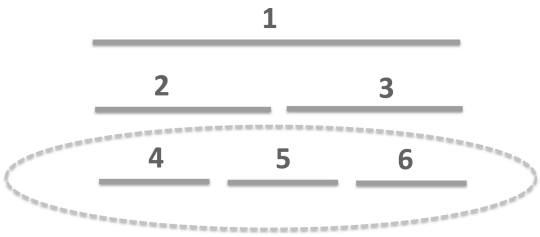
We've already seen, for the dinner puzzle (puzzle 8), the talent puzzle (puzzle 9), and the making-change-with-the-fewest-bills problem (exercise 3 of puzzle 15), that greed doesn't lead to optimal solutions. But since greed comes naturally to most humans, we'll continue to explore the greedy approach and see how it works out for us in this puzzle!

Our problem is to maximize the number of courses that a student can take in any given semester. Since the student wants to graduate in a minimum number of semesters, the student would like to take as many courses as possible.[1] Since attendance is mandatory, the student needs to choose classes that do not have scheduling conflicts.

The student is given the entire course schedule as a list of intervals. Each interval is of the form [*a*, *b*), where *a* and *b* are hours in the day, with $a < b$. The square bracket means the interval is closed at the beginning, and the curved

bracket means the interval is open at the end. This means the student can take two courses that meet at [*a*, *b*) and [*b*, *c*). For example, professors at MIT are supposed to finish five minutes before the hour so students have ample time to get to the next class, which is supposed to start five minutes past the hour. We saw this notion of intervals previously in the celebrity puzzle (puzzle 2).

Our problem, given a list of intervals, is to choose a subset of nonintersecting intervals that has maximum size. Intervals lend themselves to nice pictorial representation, so let's look at a few examples in pictures.



In this example, if we select course 1, that is the only course we can take. Or we could select courses 2 and 3. The maximum number of courses we can select without a scheduling conflict is three: courses 4, 5, and 6.

A Greedy Approach

We will use a greedy approach with the following structure.

. Select a course *c* using some rule.
. Reject all courses that conflict with *c*.
. Go to step 1 if the set of courses is not empty.

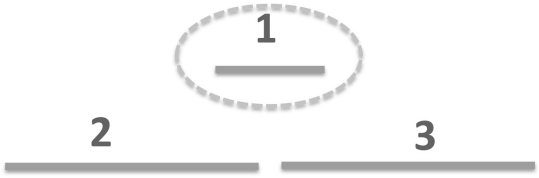
If we decided to select the course with the smallest number (course 1), that clearly would not work for this example.

Shortest Duration Rule

In this example, if we select the shortest-duration course, say course 4, then we cannot take courses 1 and 2. If we select the shortest course from the ones remaining, we select course 5, which eliminates course 3. That leaves us with course 6, which we select to obtain the maximum number of courses.

Will repeatedly selecting the shortest course work in all cases? If not, what rule should we use to ensure a maximum number of courses is selected for all possible course schedules?

If you convinced yourself that the shortest duration rule will work in all cases —surprise! Here's a simple example where picking the shortest course in step 1 does not work.

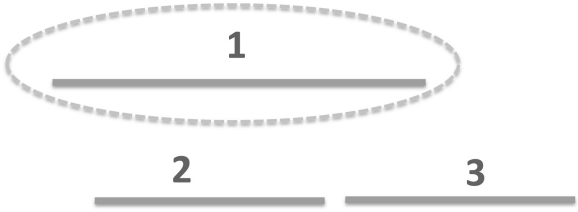


Course 1 is the shortest course, but if the student takes it, neither of the other two courses (2 and 3)—which do not conflict with each other—can be taken.

Earliest Start Time Rule

How about a rule that chooses the course with the earliest start time? This way, the student can wake up bright and early and have a packed schedule for the entire day. The earliest start time works for the two examples we have so far. In the first example, course 4 starts the earliest (along with other courses), and picking it would eliminate courses 1 and 2, leaving course 5 with the earliest start time. This will lead us to the optimum selection. In the second example, course 2 starts the earliest, and picking it—followed by course 3—produces the optimum.

Unfortunately, the earliest start time rule isn't perfect either. The example below shows that it doesn't always work.



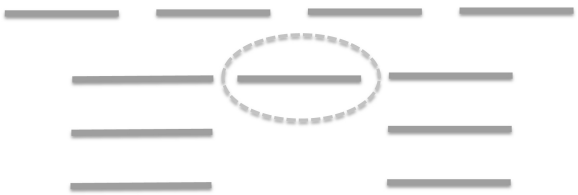
The rule selects course 1 and the algorithm ends with a one-course selection, when a two-course selection is possible.

Fewest Conflicts Rule

You might ask, "Why are we messing around with start times and durations?" We can determine if two courses conflict or not. Suppose we determine how many courses each course conflicts with, and select the course that conflicts with

the fewest. This is our rule to repeatedly apply. In the first example, course 4 conflicts with the fewest courses (two) and we will start with that, getting a three-course selection. In the second example, courses 2 and 3 conflict with the fewest courses (one each) and we will select one of those, eliminate course 1 in step 2 of the algorithm, and obtain an optimal two-course selection. Same deal in our third example. This is looking pretty good!

Unfortunately, our hopes are dashed with the following example. This example is much more involved and admittedly somewhat contrived, indicating that the fewest conflicts rule works in most cases.



In the above example, the circled course (we have omitted numbers) has two conflicts, with the two courses above it. All other courses have at least three conflicts. This means we pick the circled course. In step 2 of the algorithm, we eliminate the two courses above the circled course. This means we miss the four-course selection, corresponding to the top line, which is the optimal selection.

By now, you may think that a greedy approach to this problem will never work, and that we need some sort of exhaustive search like we had for the dinner invitation puzzle (puzzle 8) or the talent search puzzle (puzzle 9). But amazingly, there is a simple rule that works in all cases.

Earliest Finish Time Rule

We will pick the course with the earliest *finish* time. This rule works in all our examples—check it out. In the last example, which "broke" the fewest conflicts rule, we pick the leftmost course on the top line, followed by the one next to it on the same line, and so on, to obtain the four-course optimal selection. Of course, this does not prove that this rule will always find the optimal solution. Proof by example isn't a valid technique!

You have several choices at this point: (1) Try very hard to come up with an example that breaks the earliest finish time rule, and when you fail, be convinced the rule works; (2) read the proof (below), which requires familiarity with proof techniques, including proof by induction; or (3) just believe us.

Notation: $s(i)$ start time, $f(i)$ finish time, $s(i) < f(i)$ (start time must be less than finish time for a course). Two courses i and j are compatible if they don't overlap, that is, $f(i) \leq s(j)$ or $f(j) \leq s(i)$.

Claim 1. The greedy approach outputs a list of intervals $[s(i_1), f(i_1)]$, $[s(i_2), f(i_2)]$, …, $[s(i_k), f(i_k)]$ such that $s(i_1) < f(i_1) \leq s(i_2) < f(i_2) \leq \ldots \leq s(i_k) < f(i_k)$.

Proof. Proof by contradiction—if $f(i_j) > s(i_{j+1})$, intervals j and $j + 1$ intersect, which is a contradiction of step 2 of the algorithm.

Claim 2. Given a list of intervals L, a greedy algorithm with earliest finish time produces k^* intervals, where k^* is optimal.

Proof. Induction on k^*.

Base case: $k^* = 1$. The optimal number of intervals is one; that means that all intervals conflict with each other. Then, any interval works.

Inductive step: Suppose the claim holds for k^* and we are given a list of intervals whose optimal schedule has $k^* + 1$ intervals, namely:

$S^*[1, 2, \ldots, k^* + 1] = [s(j_1), f(j_1)], \ldots, [s(j_{k^*+1}), f(j_{k^*+1})]$.

Say for some generic n, the greedy algorithm gives a list of intervals:

$S[1, 2, \ldots, n] = [s(i_1), f(i_1)], \ldots, [s(i_n), f(i_n)]$.

By construction, we know that $f(i_1) \leq f(j_1)$, since the greedy algorithm picks the earliest finish time. Now we can create a schedule

$S^{**} = [s(i_1), f(i_1)], [s(j_2), f(j_2)], \ldots, [s(j_{k^*+1}), f(j_{k^*+1})]$

since the interval $[s(i_1), f(i_1)]$ does not overlap with the interval $[s(j_2), f(j_2)]$ or any intervals that come after that. Note that, since the length of S^{**} is $k^* + 1$, this schedule is also optimal.

Now we proceed to define L' as the set of intervals with $s(i) \geq f(i_1)$. Since S^{**} is optimal for L, $S^{**}[2, 3, \ldots, k^* + 1]$ is optimal for L', which implies that the optimal schedule for L' has size k^*.

We now see by our initial inductive hypothesis that running the greedy algorithm on L' should produce a schedule of size k^*. Hence, by our construction, running the greedy algorithm on L' gives us $S[2, \ldots, n]$.

This means $n - 1 = k^*$ or $n = k^* + 1$, which implies that $S[1, \ldots, n]$ is indeed optimal, and we are done.

Armed with the earliest finish time rule, we are now ready to begin coding the

greedy algorithm. Since this is a book on programming, we are going to code not just the earliest finish time rule, but two of the other rules as well. We'll show you a new programming trick along the way.

The code follows the algorithmic structure we provided at the beginning: We will have functions for the different rules, a function to eliminate conflicting courses, and a main function that invokes the first two functions repeatedly to come up with a course selection.

We'll start with the main function, shown below.

```
1.    def executeSchedule(courses, selectionRule):
2.        selectedCourses = []
3.        while len(courses) > 0:
4.            selCourse = selectionRule(courses)
5.            selectedCourses.append(selCourse)
6.            courses = removeConflictingCourses(selCourse, courses)
7.        return selectedCourses
```

We start by initializing the selected courses list to be empty (line 2). The **while** loop executes the greedy algorithm. Line 4 shows a programming construct that we have not seen before—the selectionRule argument to the function executeSchedule is itself a function. This is pretty cool, because we do not have to change the executeSchedule function at all to run it with the different rules we explored! (We do have to code each of the rules in functions with different names, and we will get to that later.)

Once we have selected the course according to selectionRule (line 4), we add it to the selected course list (line 5), then eliminate all courses that conflict with this course, and eliminate the selected course from the list courses. The **while** loop terminates when the course list goes empty.

Let's take a look at how conflicts are detected and courses eliminated in function removeConflictingCourses.

```
1.    def removeConflictingCourses(selCourse, courses):
2.        nonConflictingCourses = []
3.        for s in courses:
4.            if s[1] <= selCourse[0] or s[0] >= selCourse[1]:
5.                nonConflictingCourses.append(s)
6.        return nonConflictingCourses
```

This procedure returns a list of courses nonConflictingCourses that do not conflict with the argument selCourse. selCourse is in the list courses when the procedure is invoked, but because selCourse conflicts with itself, it will not be in

nonConflictingCourses.

Line 4 is the most interesting line—it determines whether a course s conflicts with selCourse. Each course is represented as an interval [*a*, *b*), as we described earlier. Course s is represented as [s[0], s[1]]. If the finish time of s is less than or equal to the start time of selCourse, or the start time of s is greater than or equal to the finish time of selCourse, the two courses do not conflict and s can be appended to nonConflictingCourses (line 5). Otherwise, they do conflict, and we do not append.

Now let's take a look at the implementation of the different rules.

```
1.      def shortDuration(courses):
2.          shortDuration = courses[0]
3.          for s in courses:
4.              if s[1] - s[0] < shortDuration[1] - shortDuration[0]:
5.                  shortDuration = s
6.          return shortDuration
```

The function shortDuration assumes that the argument courses is not empty. This is ensured by the **while** loop in function executeSchedule (line 3 of that function). If you pass an empty list, the function will crash[2] on line 2. We go through the list of courses to find the one with the shortest duration.

Let's now look at the least conflicts rule implementation. This is a little more involved.

```
 1.     def leastConflicts(courses):
 2.         conflictTotal = []
 3.         for i in courses:
 4.             conflictList = []
 5.             for j in courses:
 6.                 if i == j or i[1] <= j[0] or i[0] <= j[1]:
 7.                     continue
 8.                 conflictList.append(courses.index(j))
 9.             conflictTotal.append(conflictList)
10.         leastConflict = min(conflictTotal, key=len)
11.         leastConflictCourse = \
11a.            courses[conflictTotal.index(leastConflict)]
12.         return leastConflictCourse
```

This function creates a list conflictTotal, where conflictTotal[i] corresponds to a list of courses that conflict with course i. Each conflictTotal[i] entry is constructed using

the list conflictList. We use a doubly nested **for** loop structure (lines 3–9). Our conflict check is on line 6, and is similar to the check on line 4 in RemoveConflictingCourses. We do not include course i in course i's conflict list. Since the **for** loops iterate over courses i and j in the list courses, on line 8 we find the index of the course j that conflicts with i and append it to conflictList. After each iteration of the inner **for** loop we append the current conflictList to conflictTotal (line 9).

On line 10 we use the built-in function **min** to find the element leastConflict in conflictTotal that has the minimum length, meaning it is the conflict list containing the minimum number of course indices. The call to conflictTotal.index(leastConflict) produces the index of leastConflict, which we use to index into the list courses and find the course we want.

Finally, here's code for the rule that you want to apply in practice. We are keeping up the pretense that maximum course scheduling is a practical problem that students face!

```
1.  def earliestFinishTime(courses):
2.      earliestFinishTime = courses[0]
3.      for i in courses:
4.          if i[1] < earliestFinishTime[1]:
5.              earliestFinishTime = i
6.      return earliestFinishTime
```

This code is similar to the code for the shortest duration rule, and also assumes that the argument list is not empty. On lines 4 and 5 we use the endpoints of the course intervals (i.e., the finish times) to find the course with the earliest finish time.

To run the algorithm with an appropriate rule, we obviously need a list of courses, and then we simply invoke executeSchedule with appropriate arguments, as shown below.

If we run:

```
mycourses = [[8,9], [8,10], [12,13], [16,17], [18,19],
             [19,20], [18,20], [17,19], [13,20],
             [9,11], [11,12], [15,17]]
print ('Shortest duration:', executeSchedule(mycourses, shortDuration))
print ('Earliest finish time:', executeSchedule(mycourses, earliestFinishTime))
```

We get:

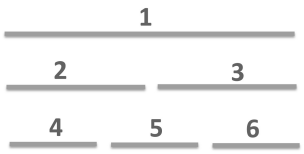
```
Shortest duration: [[8, 9], [12, 13], [16, 17], [18, 19],
```

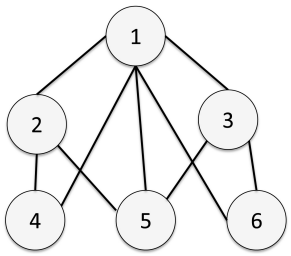
[19, 20], [11, 12], [9, 11]]
Earliest finish time: [[8, 9], [9, 11], [11, 12], [12, 13],
[16, 17], [18, 19], [19, 20]]

In this example, the two rules produce the same set of courses, though the selection happens in a different order.

When Is Greed Good?

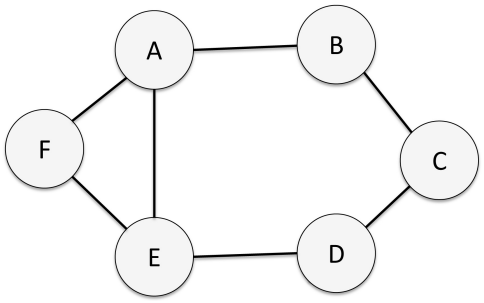
Suppose we construct a graph from a given set of intervals. Each course corresponds to a vertex, and there is an edge between each pair of vertices/courses whose intervals intersect. Here's a course schedule and the corresponding graph.





Our goal in this puzzle was to select a maximum set of vertices that do not have edges between them (i.e., the corresponding courses do not conflict). This is exactly like our dinner puzzle (puzzle 8) and the maximum independent set problem. And we have proven that a greedy algorithm produces the optimal result (i.e., selects the maximum number of courses) in all cases. Are we ready to claim we solved a Millennium Prize Problem?

Nope. The problems are not completely equivalent. The graph that is produced from a set of course intervals is a special type of graph called an *interval graph*. The maximum independent set problem for interval graphs admits an efficient solution. But not all graphs are interval graphs. For example, the graph below—which you saw in puzzle 8—is not an interval graph. In other words, it *cannot* be generated from any list of intervals.



The above graph could obviously correspond to dislike relationships, which are subject to human whims. The reason it is not an interval graph is that it has a cycle of length equal to or greater than 4 without "shortcuts," namely, *A*—*B*—*C*—*D*—*E*—*A*. A cycle is simply a path of edges in a graph that begins and ends with the same vertex. In our example, that vertex is *A*. Recall that the greedy approach of selecting a vertex with the fewest connecting edges fails for this example. And no one has been able to discover a greedy approach using any selection rule that produces the optimal result for *arbitrary* graphs. Our exhaustive solutions to the dinner puzzle continue to be relevant.

We have seen greedy algorithms fail for the problem in puzzle 9 and the problem of making change with the fewest bills (exercise 3 in puzzle 15). The most famous problem for which a greedy algorithm produces the optimum solution for all problem instances, is the shortest path problem on graphs where the edge weights are nonnegative integers. The greedy algorithm is called Dijkstra's algorithm after its inventor, Edsger Dijkstra. The shortest path problem is related to puzzle 20, "Six Degrees of Separation."

Exercises

Exercise 1: One poor student is stuck taking the maximum number of classes possible, but the student wishes to attend the minimum amount of class time so there is maybe still time for a social life. Modify the earliest finish time selection function so in case of ties it selects the course with the *shortest* duration. Given the following set of courses,

```
scourses = [[8, 10], [9, 10], [10, 12], [11, 12],
            [12, 14], [13, 14]]
```

invoking the new rule in executeSchedule(scourses, NewRule) should return

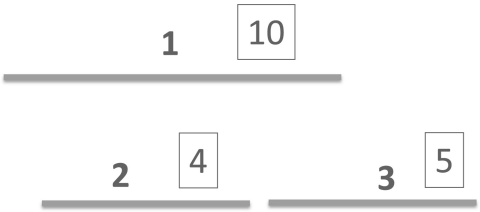
```
[[9, 10], [11, 12], [13, 14]]
```

as opposed to the output of executeSchedule(scourses, EarliestFinishTime):

```
[[8, 10], [10, 12], [12, 14]]
```

Our fictitious student will be indebted to you.

Puzzle Exercise 2: Suppose courses have weights. Our student's goal is not to take the maximum number of courses that do not conflict, but rather, to take a collection of nonconflicting courses with maximum weight. Does the earliest finish time rule produce the maximum weight selection in all cases? Unfortunately, it does not, as the following simple example illustrates.



In this example, the earliest finish time rule will select courses 2 and 3, with a weight of 9, whereas the maximum weight selection is the single course 1, with weight 10.

Finding the maximum weight selection of nonconflicting classes requires a more sophisticated algorithm. One way is to generate all possible combinations of courses (like we did with guests in puzzle 8), eliminate each combination that contains any pair of conflicting courses, and select the maximum weight combination from the remaining "good" combinations. For *n* courses, this will mean generating 2^n combinations. A better way is the following strategy, given in pseudocode:

recursiveSelect(*courses*)

Base case: Do nothing if *courses* is empty.

For each course *c* in *courses*:

Later courses = courses that start at or after *c* finishes.

Selection = *c* + recursive selection from later courses.

Keep maximum weight selection seen thus far.

Return maximum weight selection.

Code a solution based on the above pseudocode. Consider the following example, where the third number is the weight of the course:

```
wcourses = [[8, 10, 1], [9, 12, 3], [11, 12, 1],
            [12, 13, 1], [15, 16, 1], [16, 17, 1],
            [18, 20, 2], [17, 19, 2], [13, 20, 7]]
```

You should get the course selection:

```
[[9, 12, 3], [12, 13, 1], [13, 20, 7]]
```

with a weight of 11—the maximum possible.

You might wonder why we are only considering courses that start at or after the chosen course finish time, rather than looking at all courses that do not conflict with the chosen course. We are looking at all possible "starting" courses in the *for* loop. This means that we can eliminate repetition by not generating a solution corresponding to course 2 followed by course 1, after generating the solution course 1 followed by course 2. This makes the algorithm significantly more efficient than an algorithm that considers all non-conflicting courses. A related optimization was performed in puzzle 15 to avoid repeated solutions in counting change.

Puzzle Exercise 3: Rather than maximizing the number of nonconflicting courses that a student can take, a student wishes to maximize the amount of time that he or she attends class.[3] Rules: The student can only register for a nonconflicting schedule of classes, can only attend classes that he or she is registered for, and needs to attend the entire lecture for each registered class. Sitting in an empty lecture hall doesn't count!

Write code that optimally solves this maximum class attendance time problem for any course schedule.

Hint: Think of what the weights of each class should correspond to.

Notes

1. We are definitely not recommending this course of action. This puzzle is totally made up!

2. To be precise, it will throw a "list index out of range" exception since the list element does not exist. We will look at exceptions in puzzle 18.

3. This student is the masochistic version of the student from exercise 1.

17

Anagramania

"the best things in life are free" → "nail biting refreshes the feet"
—Donald L. Holmes

> Programming constructs and algorithmic paradigms covered in this puzzle: Dictionary basics. Hashing.

An anagram is a word, phrase, or name formed by rearranging the letters of another, such as *cinema*, formed from *iceman*. Let's say we have a large corpus of words and our job is to group all the anagrams together. That is, we need to split the corpus into some number of groups, where each group contains words that are anagrams of each other. One approach is to sort the corpus, so words that are anagrams of each other are placed next to each other. Given:

ate but eat tub tea

We produce:

ate eat tea but tub

Or more interestingly, suppose we are given:

abed abet abets abut acme acre acres actors actress airmen alert alerted ales aligned allergy alter altered amen anew angel angle antler apt bade baste bead beast beat beats beta betas came care cares casters castor costar dealing gallery glean largely later leading learnt leas mace mane marine mean name pat race races recasts regally related remain rental sale scare seal tabu tap treadle tuba wane wean

How do we proceed?

Finding Anagram Groupings One at a Time

We could use the following strategy:

For each word *v* in *list*:
  For each word *w* ≠ *v* in *list*:
    Check if *v* and *w* are anagrams.
    If so, move *w* next to *v*.

The code for this is shown below. It works, but is quite inefficient, being a doubly nested loop of checks—and the check itself invokes a sort procedure.

```
1.  def anagramGrouping(input):
2.      output = []
3.      seen = [False] * len(input)
4.      for i in range(len(input)):
5.          if seen[i]:
6.              continue
7.          output.append(input[i])
8.          seen[i] = True
9.          for j in range(i + 1, len(input)):
10.             if not seen[j] and anagram(input[i], input[j]):
11.                 output.append(input[j])
12.                 seen[j] = True
13.     return output
```

The code closely follows the pseudocode. It takes a list of strings as input, for instance, ['ate', 'but', 'eat', 'tub', 'tea'], and produces another list as output, ['ate', 'eat', 'tea', 'but', 'tub']. Line 3 declares a list variable, seen, that is the length of the input corpus, and initializes all its entries to **False**. The variable seen keeps track of words that have already been put into the output list output. In the outer **for** loop, we pick a word that has not already been put into output, put it into output, and look for words that are anagrams of this chosen word. This word will be at the front of its anagram grouping. Filling out the anagram grouping requires the inner **for** loop beginning on line 9. Line 10 checks whether the word is already in output (seen[j] should be **False**) and whether it is an anagram of the chosen word. If so, it is appended to the output list in the anagram group for the chosen word.

Each inner **for** loop produces one anagram grouping.

The check **not** seen[j] on line 10 is not really necessary. Let's say that v has just been put on the output list and we are trying to find the anagrams of v. Any word w that appears after v in the input list, and has already been put on the output list, is an anagram of a different word that was processed before v. This means that w cannot be an anagram of v and the function call anagram(v, w) will return **False**.

So why perform the **not** seen[j] check? This is a small performance optimization—if seen[j] is **True**, the **if** statement will immediately return **False** and anagram will not be called. Calling anagram is more expensive than merely checking if a variable is **True**, as will become clear below.

Here's code that checks if two words (strings) are anagrams of each other.

```
1.  def anagram(str1, str2):
2.      return sorted(str1) == sorted(str2)
```

Line 2 simply invokes Python's built-in sorting function, which sorts the letters in the strings in lexicographic order and returns a list of characters. sorted('actress') produces ['a', 'c', 'e', 'r', 's', 's', 't'] and sorted('casters') produces ['a', 'c', 'e', 'r', 's', 's', 't'] as well, since 'actress' and 'casters' are anagrams. Words that are not anagrams of each other will fail this test since the (sorted) lists of characters will be different.

It doesn't matter what order you print the anagram groups in, as long as all anagram groups are unbroken. And the order within each group can be arbitrary. All these outputs are valid.

ate	eat	tea	but	tub
ate	tea	eat	but	tub
but	tub	ate	eat	tea
tub	but	eat	ate	tea

If we have a list of words of length n, and on average each word has, say, m letters, we are doing roughly $n^2/2$ anagram checks. The 1/2 factor is because we are comparing each distinct pair of words, and don't compare v with w if we have already compared w with v. Each anagram check requires roughly $2m \log m$ comparisons of characters, given that we are sorting m letters in each of two words. So we are doing $n^2 m \log m$ comparisons in total.

Can you think of a much more efficient way of taking an arbitrary list of words and producing a new list where the anagrams are all grouped together?

Anagram Grouping via Sorting

We found a representation of a word s using sorted(s) that is canonical, in the

sense that anagrams have the same representation. However, we did something inefficient in anagramGrouping by using a doubly nested loop to group anagrams together. Suppose we pair each word with its canonical representation. That is, we create 2-tuples of the form (sorted(s), s). The first item is a list of characters and the second item is a string. In our small corpus ['ate', 'but', 'eat', 'tub', 'tea'] we will get the following five tuples:

```
(['a', 'e', 't'], 'ate')
(['b', 't', 'u'], 'but')
(['a', 'e', 't'], 'eat')
(['b', 't', 'u'], 'tub')
(['a', 'e', 't'], 'tea')
```

Now, what happens if we sort the tuples in ascending order? The default comparison in Python's built-in sorting uses lexicographic order and goes from left to right in each tuple. Upon sorting, we will get:

```
(['a', 'e', 't'], 'ate')
(['a', 'e', 't'], 'eat')
(['a', 'e', 't'], 'tea')
(['b', 't', 'u'], 'but')
(['b', 't', 'u'], 'tub')
```

First, the lists of characters are sorted lexicographically, meaning that all the anagrams are grouped together since they have identical lists of characters. And the lists that start with 'a' come before the lists that start with 'b'. Finally, the words associated with each anagram are sorted lexicographically within a given anagram group.

Here's code that implements the above algorithm.

```
1.  def anagramSortChar(input):
2.      canonical = []
3.      for i in range(len(input)):
4.          canonical.append((sorted(input[i]), input[i]))
5.      canonical.sort()
6.      output = []
7.      for t in canonical:
8.          output.append(t[1])
9.      return output
```

Lines 2–4 construct the list of 2-tuples. sorted(input[i]) must be the first item in each tuple for this algorithm to work. (If you don't know why, see what happens when you switch the order.)

Line 5 sorts the list canonical in-place, that is, the list is mutated. There is no

need for us to keep a copy of the unsorted list. Lines 6–8 produce a new output list by discarding the first item of each tuple from canonical. They have served their purpose in producing the anagram grouping via sorting.

Let's assume again that we have a list of words of length n, and each word has, on average, m letters. We are sorting the characters in each word, which takes a total of $nm \log m$ comparisons. Then, we are sorting the n tuples, and that takes $n \log n$ comparisons of tuples. If we think of each tuple comparison as requiring roughly m comparisons of characters, then the number of comparisons of characters in the second step is $mn \log n$. So the overall number of comparisons is $nm(\log m + \log n)$. This is much better than the $n^2 m \log m$ comparisons in anagramGrouping. However, the downside of anagramSortChar is that we have to store a list of 2-tuples of length n, with the first item of each 2-tuple being a list of characters. We did not have to store these lists of characters in anagramGrouping.

Anagram Grouping via Hashing

There is an even more efficient strategy that does not require sorting the characters of each word, and storing this sorted representation for each word, prior to sorting the entire list of words. This method uses the concept of *hashing*. (As an aside, hashing is central to the dictionary data structure of Python.)

We can compute the *hash* of a string by assigning a unique number to each character and computing some function over these numbers. Typically, this function is multiplication.

```
hash('ate') = h('a') * h('t') * h('e') = 2 * 71 * 11 = 1562
hash('eat') = 1562
hash('tea') = 1562
```

For this hash function, all anagrams will definitely have the same hash. So if we sort the words in the corpus based on what the hash values are, all the anagrams should be grouped together in the sorted corpus. We still have one problem, though—two words that are not anagrams may end up with the same hash. For example, if h('m') happens to be 781, the word 'am' will also have a hash of 1562. The word 'am' may appear in between 'ate' and 'eat' in our sorted corpus.

Luckily, this problem is easily solved. We will use prime numbers (as we did above) for the hash values corresponding to each letter. Given that each number has a unique prime factorization, the above problem will not occur. Note that this disallows h('m') being 781 because 781 = 11 * 71 is not a prime.

Let’s elaborate. Recall that the unique prime factorization theorem states that every integer greater than 1 is either prime itself or is the product of prime numbers, and that this product is unique, up to the order of the factors. If we assign unique primes (uppercase letters used as symbols to represent prime numbers) to each letter of the alphabet (lowercase letters), then 'altered' can be represented as the number A•L•T•E²•R•D. The word 'alerted' is represented as A•L•E²•R•T•D, which is obviously the same number. And thanks to the unique prime factorization theorem, we can only get this number if we have a word with exactly one a, one l, two e’s, one r, one t, and one d in it, which would be an anagram of 'altered' and 'alerted'.

To summarize, the efficient strategy to solve the anagram puzzle is simply to compute a hash of each word by assigning a unique prime to each letter of the alphabet and taking the product. We then sort the words based on the hashes, resulting in all anagrams being grouped together in the sorted output. We’ll look at compact code that solves our problem after we take a small digression to describe dictionaries in Python.

Dictionaries

Lists have to be indexed using nonnegative integers, but dictionaries in Python are generalized lists that can be indexed using strings, integers, floating-point numbers (floats), or tuples. You will appreciate the power of dictionaries in this and the next few puzzles.

Here’s a simple dictionary that maps names to IDs. Note the curly braces, which tell you that we are declaring a dictionary.

```
NameToID = {'Alice': 23, 'Bob': 31, 'Dave': 5, 'John': 7}
```

NameToID['Alice'] returns 23, and NameToID['Dave'] returns 5. NameToID['David'] throws an error. The indices of our generalized list are called keys, and the above dictionary has four keys. Each key in a dictionary points to a value, so a dictionary is composed of key-value pairs.

While NameToID['David'] throws an error, we can check if a key exists in a dictionary by writing:

```
'David' in NameToID
'David' not in NameToID
```

This will return **False** and **True**, respectively, for our dictionary. However, if we write:

```
NameToID['David'] = 24
print(NameToID)
```

We will see:

```
{'John': 7, 'Bob': 31, 'David': 24, 'Alice': 23, 'Dave': 5}
```

There are now five key-value pairs. If we write:

```
'David' in NameToID
```

This will now return **True** since we added 'David' as a key to NameToID.

Notice that the keys printed in a different order this time. Dictionaries in Python do not guarantee any particular ordering of key-value pairs. Unlike a list, where the indices are nonnegative numbers and have a natural ordering, in a dictionary, keys can be integers, strings, or tuples without a natural order. Here's a more interesting example of a dictionary:

```
crazyPairs = {-1: 'Bob', 'Bob': -1, 'Alice': (23, 11), (23, 11): 'Alice'}
```

We have a crazy collection of "objects," which are numbers, tuples, or people, and we have paired them up in our dictionary. We can add another mapping, for instance, crazyPairs['David'] = 24, or change an existing mapping by writing crazyPairs['Alice'] = (23, 12). If we do this, and execute print(crazyPairs), we get:

```
{(23, 11): 'Alice', 'Bob': -1, 'David': 24,
'Alice': (23, 12), -1: 'Bob'}
```

Notice that that the key-value pair (23, 11): 'Alice' was unaffected since we only modified values associated with other keys.

We were careful to use immutable tuples in our example above. Lists are not allowed as keys in a dictionary in Python. The reason is that lists are mutable, and all sorts of weird bugs arise if you insert a list as a key into a dictionary, then modify the list. This will "confuse" the dictionary, and Python precludes this confusion by not allowing lists to be used as keys. Lists can, however, be used as values in a dictionary, and can be mutated after insertion into the dictionary.

Finally, to delete a key from a dictionary, you can write:

```
if 'Alice' in NameToID:
    del NameToID['Alice']
```

This will delete 'Alice' from NameToID if it exists in NameToID.

Given a dictionary d, to obtain lists corresponding to dictionary keys, values, and key-value pairs represented as tuples, we can call d.keys(), d.values(), and

d.items(), respectively.

We have covered the rudiments of dictionaries and their usage. You will see other operations on dictionaries in subsequent puzzles. We saw sets in the sudoku puzzle (puzzle 14). You can think of sets as dictionaries without values.

Using Dictionaries to Group Anagrams

```
1.  chToprime = {'a': 2, 'b': 3, 'c': 5, 'd': 7,
                 'e': 11, 'f': 13, 'g': 17, 'h': 19,
                 'i': 23, 'j': 29, 'k': 31, 'l': 37,
                 'm': 41, 'n': 43, 'o': 47, 'p': 53,
                 'q': 59, 'r': 61, 's': 67, 't': 71,
                 'u': 73, 'v': 79, 'w': 83, 'x': 89,
                 'y': 97, 'z': 101}

2.  def primeHash(str):
3.      if len(str) == 0:
4.          return 1
5.      else:
6.          return chToprime[str[0]] * primeHash(str[1:])

7.  sorted(corpus, key=primeHash)
```

Line 1 simply assigns each letter a prime number. The first twenty-six primes are assigned to the twenty-six letters of the alphabet. The data structure is a dictionary. So if we write chToPrime['a'], the dictionary chToPrime returns 2. Similarly, chToPrime['z'] returns 101.

In function primeHash, recursion and list slicing are conveniently used to produce the hash of the word/string. The base case is an empty string, which has hash 1. Line 6 is the workhorse, where the first letter of the current word is converted to a prime, and this prime multiplies the hash computed on the word minus the first letter.

If we run the code:

```
corpus = ['abed', 'abet', 'abets', 'abut',
'acme', 'acre', 'acres', 'actors',
'actress', 'airmen', 'alert', 'alerted',
'ales', 'aligned', 'allergy', 'alter',
'altered', 'amen', 'anew', 'angel', 'angle',
'antler', 'apt', 'bade', 'baste', 'bead',
'beast', 'beat', 'beats', 'beta', 'betas',
'came', 'care', 'cares', 'casters',
'castor', 'costar', 'dealing', 'gallery',
'glean', 'largely', 'later', 'leading',
'learnt', 'leas', 'mace', 'mane', 'marine',
```

```
'mean', 'name', 'pat', 'race', 'races',
'recasts', 'regally', 'related', 'remain',
'rental', 'sale', 'scare', 'seal', 'tabu',
'tap', 'treadle', 'tuba', 'wane', 'wean']

print (sorted(corpus, key=primeHash))
```

It produces:

```
['abed, 'bade', 'bead', 'acme', 'came',
'mace', 'abet', 'beat', 'beta',
'acre', 'care', 'race', 'apt', 'pat', 'tap',
'abut', 'tabu', 'tuba', 'amen', 'mane',
'mean', 'name', 'ales', 'leas', 'sale',
'seal', 'anew', 'wane', 'wean', 'abets',
'baste', 'beast', 'beats', 'betas', 'acres',
'cares', 'races', 'scare', 'angel', 'angle',
'glean', 'alert', 'alter', 'later',
'airmen', 'marine', 'remain', 'aligned',
'dealing', 'leading', 'actors', 'castor',
'costar', 'antler', 'learnt', 'rental',
'alerted', 'altered', 'related', 'treadle',
'actress', 'casters', 'recasts', 'allergy',
'gallery', 'largely', 'regally']
```

All the anagrams nicely grouped together!

If we had wanted to use a regular list, we could have done so by using the ord function, which you encountered earlier, in the tiling puzzle (puzzle 11). Here's an alternate implementation using ord.

```
1.  primes = [2, 3, 5, 7, 11, 13, 17, 19, 23,
                29, 31, 37, 41, 43, 47, 53, 59,
                61, 67, 71, 73, 79, 83, 89, 97, 101]

2.  def chToprimef(ch):
3.      return primes[ord(ch) - 97]

4.  def primeHashf(str):
5.      if len(str) == 0:
6.          return 1
7.      else:
8.          return chToprimef(str[0]) * primeHashf(str[1:])

9.  sorted(corpus, key=primeHashf)
```

Line 1 simply creates a list of the first twenty-six primes. We have created a chToprimef function that computes the appropriate integer index for each letter.

ord('a') is 97, and we want to access primes[0] when we encounter the letter 'a'. ord('z') is 122, and we access primes[25] when we encounter the letter 'z'.

The only modification in computing the prime hash is to call chToprimef on line 8, instead of accessing the chToprime dictionary on line 6 of the original primeHash function.

How fast does this code run? Given n items in a list, we will assume we need $n \log n$ comparisons to sort the list. If there are, on average, m letters in each word, we only need m multiplications to compute the hash of a word. Let's assume we dynamically compute the hashes of the two words being compared. So the number of operations will be $2mn \log n$ for the efficient code, as compared to $2n^2m \log m$ for anagramGrouping. Say $m = 10$ and $n = 10{,}000$. Big difference!

In comparison to anagramSortChar, the performance improvement is nowhere as significant. However, the memory requirement for anagramSortChar is substantially greater than for primeHash.

We can imagine finding the hash of each word before sorting. This hash computation takes nm operations for n words, so the total number of operations is $mn + n \log n$. For the example above, using hash precomputation won't make a discernible difference, given the speed of today's computers.

Hash Tables

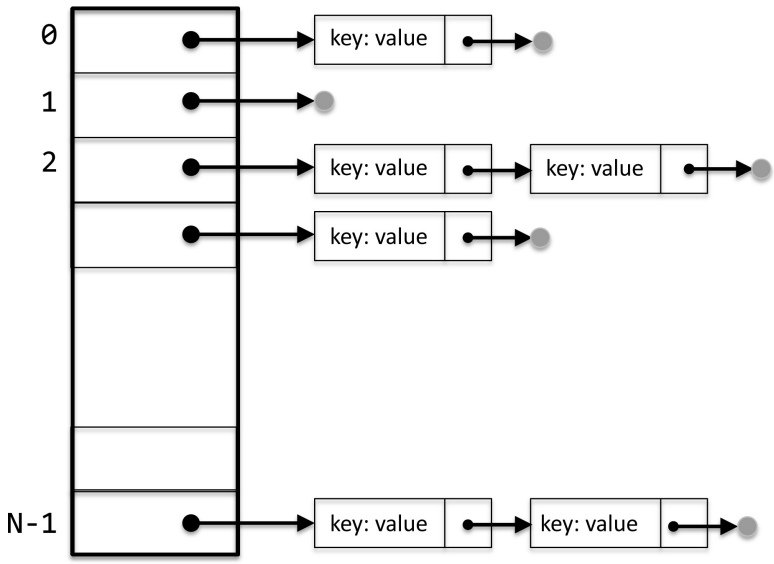
Dictionaries can be indexed using keys that are not forced to be nonnegative integers, as in lists or arrays. How are dictionaries actually implemented? In modern computers, we have to implement memory as a finite array of contiguous locations and index into these locations using nonnegative integer indices. So we have to hash each non-integer or integer key into a nonnegative integer index. This can be challenging given the range of allowed keys—keys can be rationals, strings, or tuples of strings and numbers—and because of the limits on index size.

Here's what happens inside a dictionary. Given a key, a hash function hash is applied. The function hash will first convert the key into a large integer. If you run hash('a') in Python you may get -2470766169773160532, and hash('alice') may give 4618924028089005378.[1] Even ignoring the fact that these numbers can be negative, we can't possibly have this many distinct memory locations. A common strategy is to allocate a dictionary with a much smaller number of locations $N = 2^p$ and then simply use the last p bits in the binary representation of the computed hash

to get an index between 0 and $N - 1$. The dictionary value for the key is then stored at the determined index location.

Given the huge range of possible keys, there will invariably be collisions—that is, two keys will be assigned the same index. For example, `hash('k') = 3683534172248121396` and so `'k'` will collide with `'a'` for small p, since the last few bits of both hashes are `0`'s. Collisions can be resolved in various ways—one common way is to use chaining.

Briefly, in chaining, each index location is not a single location but rather a chained list of locations. A chained hash table that implements a dictionary might look like the following diagram.



There are N index locations in the hash table. There are empty index locations (e.g., index `1`), where thus far no keys have been mapped. There are index locations with a single key (e.g., index `0`), and locations (e.g., index `2`) where collisions have occurred and multiple key-value pairs appear in the chain. When a key is looked up, we find its index location using the function `hash`, and then we may need to search the entire chain rooted at that index location to figure out if the key is in the hash table. Both the key and value are stored in items in the chain. If the stored key matches the key being looked up, the value is returned. The longer the chain, the longer this will take.

The goal of a hash table is to evenly distribute keys across the different index locations, so on average there is one key-value pair for each index location. We want even distribution for lookup efficiency. We'll explore key distribution in exercises 2 and 3 in the context of anagram grouping.

Exercises

Exercise 1: Admittedly, we "cheated" by using a built-in sorting function, but we have already shown you code for two different sorting algorithms. Modify the quicksort function to sort based on hash values obtained using chToprimef, and use it to generate the sorted corpus rather than invoking a built-in sort.

Exercise 2: We will explore the notion of hashing in this exercise and come up with yet another way of grouping anagrams together. This way is not quite foolproof, however.

Suppose we have a list L of length n = 30. Each element of L is a list of strings. You can create a list of empty lists using:

```
L = [  [ ]  for x in range(30)  ]
```

Each L[i] is an empty list, and we will add to it. We want to add all the strings in corpus to elements of L. We can take any string and convert it to a number using the primeHash function. Since the hash can be larger than the length of the list, we add each string s in corpus to the list L[primeHash(s) % p], where p = 29, the largest prime less than 30.

Write the code that does what was described above. Print out the list L. You'll notice that L will have many elements that are empty lists, but some elements will be lists containing several strings. If a word from corpus is in a particular L[i] list, all its anagrams should also be in the L[i] list.

Do you see any collisions of non-anagrams within any of the list elements L[i]? Why does this happen? Try increasing the length of the list to 100, and use a larger prime p = 97 to see if you can reduce or eliminate the collisions. Of course, if you increase the number of words in our corpus, that may increase the number of collisions for a given p.

If you complete this exercise, you will have implemented a rudimentary Python dictionary (or hash table)!

Exercise 3: In this exercise, we will use Python dictionaries to solve our anagram puzzle, extending the ideas in exercise 2. Recall that sorted(s) returns a list of sorted characters in s, and can be thought of as a canonical representation of s, in that any anagram of s produces the same list. We cannot use lists as dictionary keys, but we can convert the list returned by sorted to a tuple using **tuple**(sorted(s)) and use the immutable tuple as a dictionary key.

Write code that does the following: Take each word in a given corpus, and use the canonical tuple representation of the word as the key to a dictionary anagramDict. The value associated with each key should be a *list* of words or strings in the corpus, where each word corresponds to the canonical tuple representation. Once the dictionary is constructed, simply running **print**(anagramDict.values()) should give us the corpus with all the anagrams grouped next to each other.

Note

1. The exact computation and numbers are not important, and may vary for different Python versions.

18

Memory Serves You Well

The advantage of a bad memory is that one enjoys several times the same good things for the first time.
—Friedrich Nietzsche

> Programming constructs and algorithmic paradigms covered in this puzzle: Dictionary creation and lookup. Exceptions. Memoization in recursive search.

Here's a cute coin row game that corresponds to an optimization problem. We have a set of coins in a row, all with positive values. We need to pick a subset of the coins to maximize their sum, but we are not allowed to pick two adjacent coins.

Given:

14 3 27 4 5 15 1

You should pick 14, skip 3, pick 27, skip 4 and 5, pick 15, and skip 1. This gives a total of 56, which is optimal. Note that alternately picking and skipping coins does not work for this example (or in general). If we picked 14, 27, 5, and 1, we would only get 47. And if we picked 3, 4, and 15, we would get a pathetic score of 22.

Can you find the maximum value for the coin row problem below?

3 15 17 23 11 3 4 5 17 23 34 17 18 14 12 15

Obviously, our goal is a general-purpose algorithm that we can code and run to find the optimal selection. We will first use recursive search to solve this problem—we will recur on different choices, picking a coin or skipping it. If we skip a coin, we have the choice of either picking or skipping the next coin. But if we pick a coin, we are forced to skip the next coin. We return the maximum value returned by the recursive calls corresponding to the different choices.

Recursive Solution

Here's code that recursively solves the coin row problem.

```
1.  def coins(row, table):
2.      if len(row) == 0:
3.          table[0] = 0
4.          return 0, table
5.      elif len(row) == 1:
6.          table[1] = row[0]
7.          return row[0], table
8.      pick = coins(row[2:], table)[0] + row[0]
9.      skip = coins(row[1:], table)[0]
10.     result = max(pick, skip)
11.     table[len(row)] = result
12.     return result, table
```

The procedure takes a coin row as input, which is assumed to be a list. It also takes a dictionary `table` as input. The dictionary will contain information about the optimal value for the original problem, as well as subproblems of the original problem. The dictionary will be empty for the initial call. It will get filled in during the recursive search and will need to be passed to the recursive calls.[1]

We have two base cases in lines 2–7. The first base case is for the empty row, in which case we simply return `0` as the maximum value, and the updated dictionary. The dictionary `table` is updated with `0` as the value for key `0` (line 3). If the row has length `1`, we can simply return the coin value as the maximum value. In the one-coin case, we update key `1` of the dictionary with the coin value (line 6).

Lines 8 and 9 make recursive calls corresponding to picking or skipping the first coin on the row, respectively. If we add the value `row[0]` to our value, then we had better not pick `row[1]`, so the recursive call on line 8 has `row[2:]` as the argument. This means that the first two elements in the row are dropped—the

first because we picked it, and the second because of the adjacency constraint. On line 9, we make a recursive call with row[1:] as argument, and without adding in the row[0] value. Because we did not pick row[0], we are allowed to pick row[1] if we want to. Since coins returns both the value result and the dictionary table on line 12, to access result we need a [0] after its invocations on lines 8 and 9. Line 10 figures out which of the recursive calls won, and uses that call's value as the result. Line 11 fills in the appropriate dictionary entry. In the general case, the key/index for the dictionary is the length of the row for which we have computed the optimal value, and the value stored for that key/index is the optimal value found for that row.

A word about how the recursion works. We are selecting or skipping coins from the front of the list. So the smaller problems associated with the shorter rows correspond to dropping the elements from the *front* of the list, or the coins to the left of the row. In our example:

```
14 3 27 4 5 15 1
```

The sublist (length 5) that coins considers is the sublist:

```
27 4 5 15 1
```

If we were only interested in the maximum value obtainable for a coin row problem, we would simply return result and we would not even need table. But we want to know what coins were picked. Suppose that someone solved the long coin row problem (our second example) and told you that the optimal value was 126, you would need quite a bit of work to verify that. You would have to solve the coin row problem yourself. The dictionary returned has the information necessary to efficiently figure out what coins were picked, and the traceback procedure we will describe shortly shows the operations required.

If we run:[2]

```
coins([14, 3, 27, 4, 5, 15, 1], table={})
```

It returns:

```
(56, {0: 0, 1: 1, 2: 15, 3: 15, 4: 19, 5: 42, 6: 42, 7: 56})
```

The first value is the optimal value, and the dictionary is printed between the curly braces as a listing of key-value pairs. For example, table[0] = 0, table[4] = 19, table[7] = 56. The dictionary is telling us not only the optimal value for the original row, which has length 7, but also the optimal value for smaller coin row problems so we can trace back the coin selection. For example, table[4] tells us

that the optimal value is 19 for the sublist corresponding to the last four elements of the list: 4, 5, 15, 1. The maximum value is obtained by picking 4 and 15.

We will now show how to use the table values to conveniently trace back what coins were picked.

Tracing Back Coin Selection

```
 1.  def traceback(row, table):
 2.      select = []
 3.      i = 0
 4.      while i < len(row):
 5.          if (table[len(row)-i] == row[i]) or \
5a.              (table[len(row)-i] == \
5b.              table[len(row)-i-2] + row[i]):
 6.              select.append(row[i])
 7.              i += 2
 8.          else:
 9.              i += 1
10.      print ('Input row = ', row)
11.      print ('Table = ', table)
12.      print ('Selected coins are', select,
                 'and sum up to', table[len(row)])
```

The procedure traceback takes both the coin row and the dictionary as input. Note that the keys of table range from 0 to **len**(row), inclusive, whereas the indices of row will, as always, range from 0 to **len**(row)-1.

The procedure works backward in looking at dictionary keys that are the largest, that is, those that store information for the longest-row problems. Line 5 is the crucial line in the procedure. To start with, focus on the second part (after the first '\') of line 5. If we are working backward from the end of the list and see two table entries table[**len**(row)-i] and table[**len**(row)-i-2], where the latter is smaller than the former by row[i], it means we have picked the coin row[i] (this would be the i+1th coin on the row). For example, suppose i = 0. Then, the last and the third to last entries of the dictionary table are compared. These correspond to two optimal solutions relating to the original problem. The last entry, table[**len**(row)], corresponds to the optimal solution for the original problem. The third to last entry, table[**len**(row)-2], corresponds to the optimal solution for the original problem with the first two elements removed. If the last entry is larger than the third to

last entry by row[0], we can argue that the optimal solution for the original problem picked the first element, row[0]. Once the first element is picked, the second element has to be skipped. If row[0] is different from row[1],[3] the only way that the optimal solution for the original problem can equal the sum of row[0] and the optimal solution for the original problem with the first two elements discarded is if the first element is picked.

Why do we have the condition table[**len**(row)-i] == row[i] in the first part of line 5? This is simply to take the corner case into account, when i = **len**(row)-1. In this case, the second part of line 5 would crash, since **len**(row)-i-2 < 0. Thanks to the first condition and the disjunctive **or**, the second part is never executed—the first condition will evaluate to **True**, since table[1] is always set to row[**len**(row)-1].

In general, if we pick row[i], we could not have picked row[i+1], so we increment i by 2 and keep going (line 7). If we did not pick row[i], we increment i by 1 and keep going (line 9).

What happens for our example? Suppose we run:

```
row = [14, 3, 27, 4, 5, 15, 1]
result, table = coins(row, {})
traceback(row, table)
```

We get:

```
Input row = [14, 3, 27, 4, 5, 15, 1]
Table = {0: 0, 1: 1, 2: 15, 3: 15, 4: 19, 5: 42, 6: 42, 7: 56}
Selected coins are [14, 27, 15] and sum up to 56
```

Since table[7] equals table[5] + row[0] (i.e., 56 = 42 + 14), we choose row[0] = 14 and increment the counter i by 2. Since table[5] equals table[3] + row[2] (i.e., 42 = 15 + 27), we pick row[2] = 27 and increment i by 2. We next check table[3], which is not equal to table[1] + row[4] (i.e., 15 ≠ 1 + 5), so we increment i by 1. table[2] equals table[0] + row[5] (i.e., 15 = 0 + 15), so we include row[5] = 15.

Recall our second coin row problem:

```
3 15 17 23 11 3 4 5 17 23 34 17 18 14 12 15
```

The optimal value for the coin row problem is 126, obtained by selecting coins 15, 23, 4, 17, 34, 18, and 15.

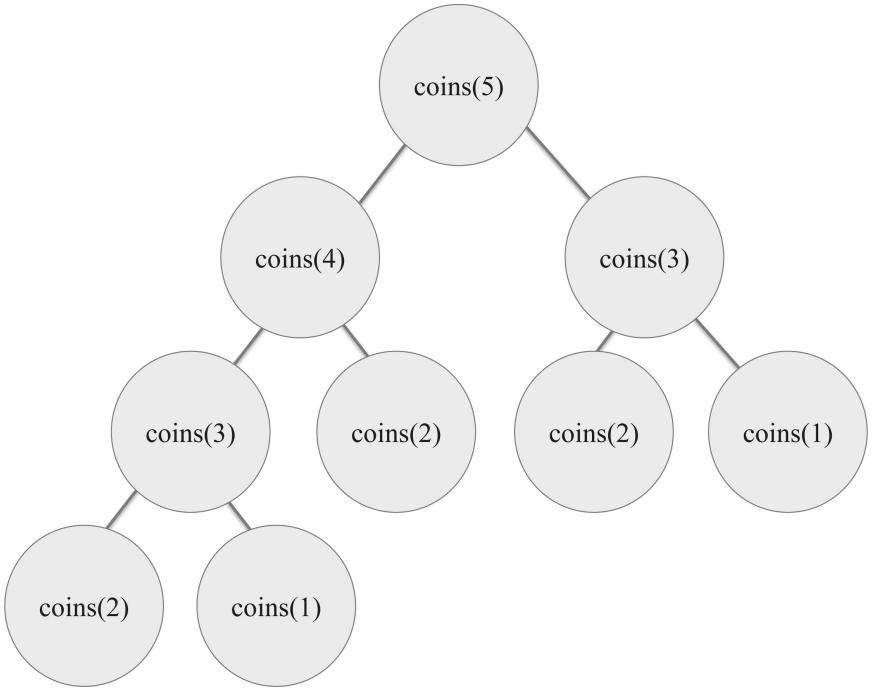
We now have an automated way of finding the optimum for an arbitrarily sized list. There is a small problem, however. We are making a lot of recursive calls, just like we did when we computed Fibonacci recursively back in the *N*-queens puzzle (puzzle 10). In fact, the number of recursive calls is exactly the same. For a list of size *n*, we end up calling the procedure with a list of size $n - 1$

and a list of size $n - 2$. Therefore, the number of calls for a list of size n is given by:

$A_n = A_{n-1} + A_{n-2}$

If $n = 40$, then $A_n = F_n = 102{,}334{,}155$. Not good.

The reason for all these calls is the redundant work that the recursive Fibonacci and the recursive `coins` functions do. The following are the recursive calls `coins` makes for a list of length 5. Not surprisingly, it looks exactly like Fibonacci. We only indicate the length of the list, since it does not matter what the list elements are to chart recursive calls.



For Fibonacci, we had an iterative solution to fall back on, which only took 40 additions to compute F_{40}. But assume we love recursion and want to use it for everything. Can we make recursive Fibonacci computation—and the recursive solution to this puzzle—more efficient? Ideally, can we reach the efficiency of iterative solutions?

Memoization

Yes, we can, through a technique called *memoization* that eliminates redundant calls. We already have a dictionary for our coin row problem, and all we need to do is look in the dictionary `table` to see if we have already computed the solution

to the problem.

Here's how memoization works in the recursive solution to our coin row problem.

```
1.  def coinsMemoize(row, memo):
2.      if len(row) == 0:
3.          memo[0] = 0
4.          return (0, memo)
5.      elif len(row) == 1:
6.          memo[1] = row[0]
7.          return (row[0], memo)
8.      try:
9.          return (memo[len(row)], memo)
10.     except KeyError:
11.         pick = coinsMemoize(row[2:], memo)[0] + row[0]
12.         skip = coinsMemoize(row[1:], memo)[0]
13.         result = max(pick, skip)
14.         memo[len(row)] = result
15.         return (result, memo)
```

We have made small but crucial modifications to the coins code. The first seven lines are exactly the same except that we have renamed table to memo. Then on line 8 we encounter, for the first time, the beginning of a Python **try**-**except** block.

The problem with looking up a list index that is out of range, or a dictionary key/index that doesn't yet exist, is that an exception is thrown. That makes sense because these entries don't exist. However, it is sometimes convenient to try these accesses and do something with the values if they are returned, and do something else if the accesses are out of bounds. This is exactly what **try**-**except** blocks allow us to do.

On line 9, we look to see if we have already computed the optimal value for the problem we are trying to solve. The optimal value is stored in memo[**len**(row)], a dictionary entry whose key is **len**(row). If the key-value pair already exists in the dictionary, we can just return it. However, if the key does not exist, we get a KeyError exception. If we did not have the **try** block, the program would crash. But thanks to the **try** block, the program control flow falls to the **except** block on line 10. In the **except** block, we know that this is the first time we are encountering this particular coin row problem. We make the recursive calls to solve the problem as in the original coins code. As before, we fill in the memo table with the computed solution (line 14). This means that the next time we encounter the

same problem, there is no need to make recursive calls.

The amazing thing is that we added three lines of code to get an exponential improvement in runtime. The memoized function only computes the solution to each problem once and stores it in the dictionary memo. There are only **len**(row) + 1 entries in memo. Each is computed exactly once, but looked up many times.

We renamed the variable table in coins to memo in coinsMemoize to reflect the different functionality of this variable. We are looking up the memo table during recursion in coinsMemoize to make the computation significantly more efficient, whereas the variable table was only written and not read in coins.

Avoiding Exceptions

We shouldn't be misled into thinking that **try-except** blocks are essential to memoization. In fact, some programmers studiously try to avoid throwing exceptions because they can be substantially more time-consuming than normal operations. Here's a small modification that avoids exceptions.

```
1.  def coinsMemoizeNoEx(row, memo):
2.          if len(row) == 0:
3.                  memo[0] = 0
4.                  return (0, memo)
5.          elif len(row) == 1:
6.                  memo[1] = row[0]
7.                  return (row[0], memo)
8.          if len(row) in memo:
9.                  return (memo[len(row)], memo)
10.         else:
11.                 pick = coinsMemoizeNoEx(row[2:], memo)[0] + row[0]
12.                 skip = coinsMemoizeNoEx(row[1:], memo)[0]
13.                 result = max(pick, skip)
14.                 memo[len(row)] = result
15.                 return (result, memo)
```

The above code performs the same checks as the exception-handling code. Realizing that fact is a good way to understand exceptions. Exceptions can be used to handle error conditions, such as dividing by zero. Wrapping many statements—each of which could fail—in a **try** block is sometimes more convenient than checking conditions before executing each statement.

Dynamic Programming

Dynamic programming is a method for solving a problem by dividing it into a collection of simpler, possibly repeated and overlapping subproblems. Dynamic programming differs from divide-and-conquer—in the latter, the subproblems are disjoint or nonoverlapping. For example, in merge sort or quicksort, the two arrays are disjoint. Similarly, in coin weighing, the coins are broken into disjoint groups. But in our coin selection example, the two subproblems are overlapping, in the sense that they have coins in common.

This overlap of subproblems means we might solve some subproblems repeatedly. In dynamic programming, each of these subproblems is solved just once, and their solutions stored. The next time the same subproblem occurs, instead of recomputing its solution, we simply look up the previously computed solution, saving computation time. Each of the subproblem solutions is indexed in some way, typically based on the values of the subproblem's input parameters, for efficient lookup. The technique of storing solutions to subproblems instead of recomputing them is therefore called "memoization."

We have used dynamic programming and memoization to efficiently solve our coin selection problem. As you will see in the exercises, the coin selection problem is not the first problem in this book amenable to dynamic programming and memoization.

Exercises

Exercise 1: Let's go back to the Fibonacci code in puzzle 10, in particular the rFib function. Use memoization to eliminate redundant calls in rFib and make it as efficient as the iterative version iFib.

One question that might be nagging you is whether we could solve the coin row problem iteratively, just like we did with Fibonacci. We absolutely can, and the code has some similarities to the code we have shown you already.

```
1.  def coinsIterative(row):
2.      table = {}
3.      table[0] = 0
4.      table[1] = row[-1]
5.      for i in range(2, len(row) + 1):
6.          skip = table[i-1][0]
7.          pick = table[i-2][0] + row[-i]
8.          result = max(pick, skip)
```

```
 9.                 table[i] = result
10.         return table[len(row)], table
```

Lines 3–4 cover the simple cases of a row with `0` length and a length of `1`. Note that these small problems correspond to the coins at the end of the list. This is why `table[1]` is filled in with `row[-1]`, which is simply the last coin on the row and is the same as writing `row[len(row)-1]`. The function `coinsIterative` produces results that are compatible with the traceback procedure and can be used instead of `coins` or `coinsMemoize`.

We could have shown you this code to begin with, explained it, and left it at that, but then you might never have learned about memoization or **try**-**except** blocks. These concepts and constructs are widely applicable and are worth having in your toolbox. Moreover, in many instances, the natural way to solve the problem is to write a recursive function. And in some of these instances, memoization can be used to ensure efficiency. For example, memoization can improve the efficiency of maximum-weight course selection in exercise 2 of puzzle 16—see exercise 3 below.

Puzzle Exercise 2: Solve a variant coin row problem: If you pick a coin, you can pick the next one, but if you pick two in a row, you have to skip *two* coins. Write recursive, recursive memoized, and iterative versions of the variant problem. As before, the goal is to maximize the value of selected coins. To obtain the selected coins, write the code to trace back the coin selection.

For our simple row example:

```
[14, 3, 27, 4, 5, 15, 1]
```

Your code should produce:

```
(61, {0: (0, 1), 1: (1, 2), 2: (16, 3), 3: (20, 3),
4: (20, 1), 5: (47, 2), 6: (47, 1), 7: (61, 2)})
```

The maximum value that can be selected, `61`, corresponds to selecting `14`, `27`, `5`, and `15`.

Hint: You will need to make three recursive calls to obey the new adjacency constraint and pick the maximum value obtained from these three calls. Instead of the two choices (pick a coin or skip a coin) in the original problem, you will have to code three choices in this variant: skip a coin, pick a coin and skip the next, or pick two adjacent coins. You may find the recursive solution easier to write first than the iterative solution.

Exercise 3: Use memoization to improve the maximum-weight course selection algorithm you wrote for exercise 2 in puzzle 16. Here's what you need to do, in pseudocode:

recursiveSelectMemoized(*courses*)
  Base case: Do nothing if *courses* is empty.
  For each course *c* in *courses*:
    Later courses = courses that start at or after *c* finishes.
    Selection = *c* + recursive selection from later courses. {Before making a recursive selection call, check memo table to see if this problem for later courses has been previously solved.}
    Keep maximum-weight selection seen thus far.
  Return maximum-weight selection after storing in memo table.

The memo table will have key-value pairs, with each key being a list of courses, and each value being the list of nonconflicting courses with maximum weight. However, Python dictionaries do not allow modifiable (or mutable) lists to be used as keys. So you will have to use repr, which converts a list into an unmodifiable (or immutable) string representation, as shown below:

```
repr([[8, 12], [13, 17]]) = '[[8, 12], [13, 17]]'
```

Then you can add to the dictionary and look in the dictionary as shown below.

```
memo[repr(courses)] = bestCourses
result = memo[repr(laterCourses)]
```

Exercise 4: We'll go back to the counting-change puzzle (puzzle 15). Your friend would like to be paid, but using the fewest number of bills. You have many different denominations and arbitrarily many bills of each denomination. In exercise 3 of puzzle 15 we asked you to solve this problem.

The code that enumerates all solutions, and the code that finds the solution with the minimum number of bills, can take a while to run when the target value is large. The minimum-bills code can be made more efficient through memoization. Memoize the code by using a memo table keyed by the *current* target value. For example, for target value 10, the memo table stores the solution corresponding to the minimum number of bills that add up to 10.

You will have to restructure makeSmartChange so that once a bill is selected and makeSmartChange is called recursively, the target value is reduced by the amount

of the selected bill. If you do this, it becomes easier to memoize based on the target value for each recursive call, once the call returns with the minimum solution for that target value.

With memoization you should be able to solve large problems and quickly find the minimum number of bills required: For example, target $1,305 with $7, $59, $71, and $97 bills. The answer is four $7, four $59, one $71, and ten $97 bills, for a total of nineteen bills.

Puzzle Exercise 5: In exercise 1 of puzzle 15, we asked you to count the number of solutions to a bill-change problem by using a global variable and modifying the provided code. Clearly, this method has the same efficiency as enumerating all distinct solutions. It shouldn't surprise you that there is a significantly more efficient way of counting the number of distinct solutions, that doesn't involve enumeration.

Suppose bills $B = b_1, b_2, \ldots, b_m$, and the target value is n. The solutions can be divided into two sets. Solutions in the first set do not contain bill b_m, and the solutions in the second set contain at least one bill b_m. We can write a recurrence:

$\text{count}(B, m, n) = \text{count}(B, m - 1, n) + \text{count}(B, m, n - b_m)$

The first argument to count is the list of bill denominations. The second argument in each count is the size of the problem (i.e., the number of bill denominations considered). That is, $m - 1$ means we are only considering $b_1, b_2, \ldots, b_{m-1}$. The third argument is the target value of the problem.

You first need to define several base cases for the above recurrence. Code recursive and recursive memoized solutions to this *real* "Counting the Ways You Can Count Change" problem based on the recurrence. Check the correctness of your solution against the inefficient enumeration solution from exercise 1, puzzle 15.

Notes

1. We could have used a list representation for `table`, in which case we would have to allocate a table with `len[row] + 1` entries beforehand. Using a dictionary will come in handy when we memoize the `coins` procedure and other recursive procedures later in the book.

2. You might think that it would have been convenient to set the default value of the dictionary `table` to `{}` in `coins` and not specify the second argument in the invocation. Python has one copy of default arguments per function. As a result, using the default argument on multiple coin row problems would result in spilling values over from the previous instance of the problem. Mutable default arguments should be used with care.

3. If row[0] is the same as row[1], it is possible that picking row[1] would also result in a solution with the same maximum value, but in either case we are guaranteed that there exists an optimal solution to the original problem that picks row[0].

19

A Weekend to Remember

Weekends don’t count unless you spend them doing something completely pointless.
—Bill Watterson

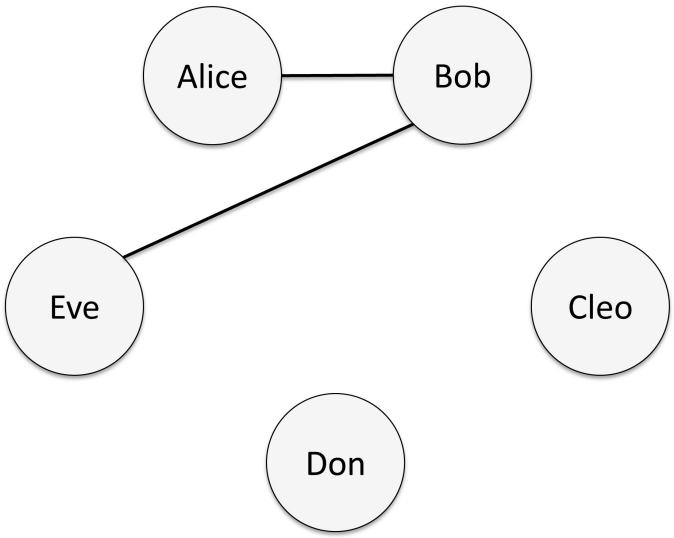
Programming constructs and algorithmic paradigms covered in this puzzle: Graph representation using dictionaries. Recursive depth-first graph traversal.

You have gotten into trouble with some of your, shall we say, difficult friends because they have realized they are not being invited to your house parties. So you announce you are going to have dinners on two consecutive nights, Friday and Saturday, and every one of your friends will be invited on exactly one of those days. You are still worried about pairs of your friends who intensely dislike each other, and you want to invite them to different days.

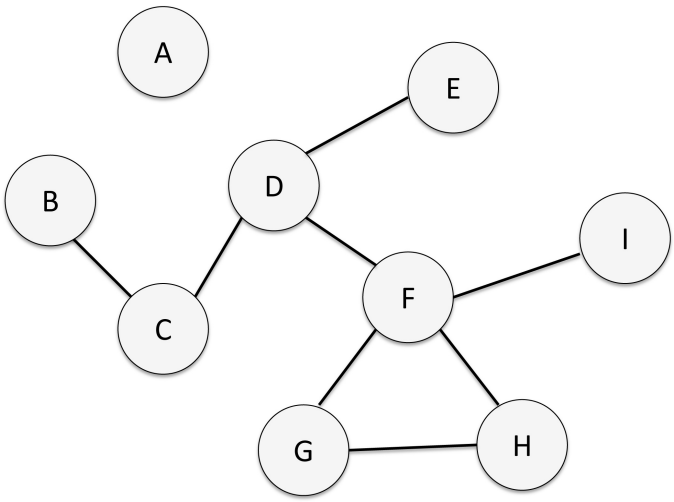
To summarize:

. Each of your friends must attend exactly one of the two dinners.
. If *A* dislikes *B* or *B* dislikes *A*, they cannot both be in the same dinner party.

Now, you are a bit worried because you don’t know if you can pull this off. If you had a small social circle like the following example, it would be easy. Recall that in this graph, the vertices are friends, and an edge between a pair of vertices mean that the two friends dislike each other and cannot be invited to the same party.



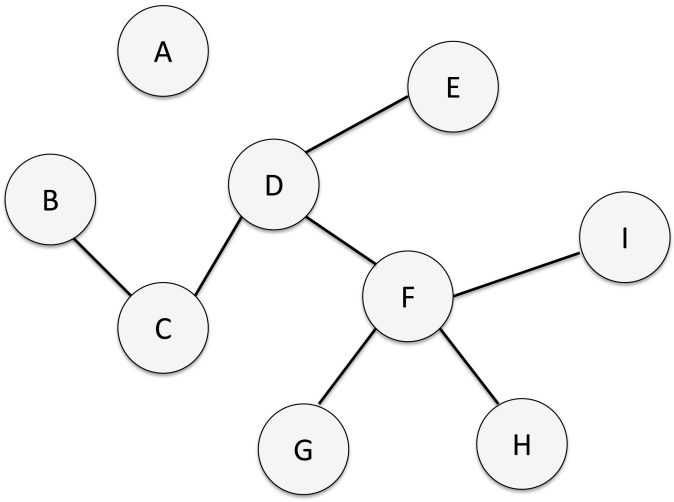
For dinner 1 you could invite Bob, Don, and Cleo. And for dinner 2 you could invite Alice and Eve. There are other possibilities as well. Unfortunately, that social circle was from many moons ago, and you have moved on. Your social circle now looks like this:



Can you invite all your friends *A* through *I* above, following your self-imposed rules? Or will you have to go back on your word, and not invite someone?

You're hosed. If you look at the graph carefully, you will see that you have three friends *F*, *G*, and *H*, each of whom can't be invited with the other two. You will need three separate days to host each of them.

Okay, let's say you convince *G* and *H* not to cause a ruckus if they see each other. So your social graph now looks like this.



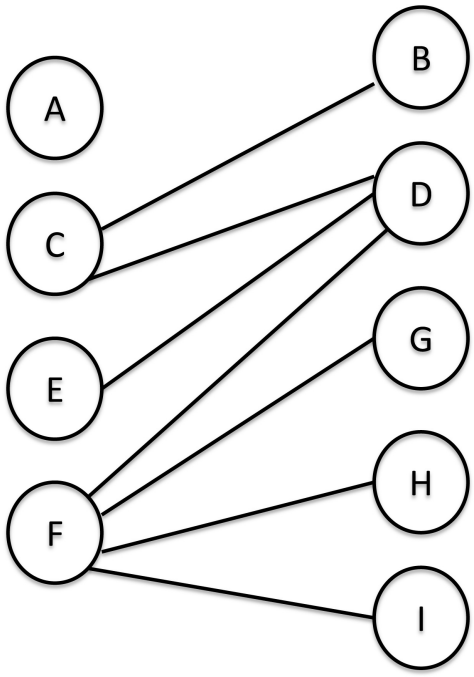
What about now?

Finding a Partition

With some work, you probably figured out that you could invite *B*, *D*, *G*, *H*, and *I* to dinner 1, and *A*, *C*, *E*, and *F* to dinner 2. Phew!

Your social circle is volatile, and you might have to do this analysis week after week. You would like to quickly know if you can make an announcement any given week, such as “party on both nights, everyone gets to come to exactly one,” with confidence that it will be a boisterous but friendly weekend. You want to write a program to check if there is some way you can “partition” (i.e., split) your friends across two nights, so that if you look at the social circle graph, all the dislike edges will be across the partition, and none within either side.

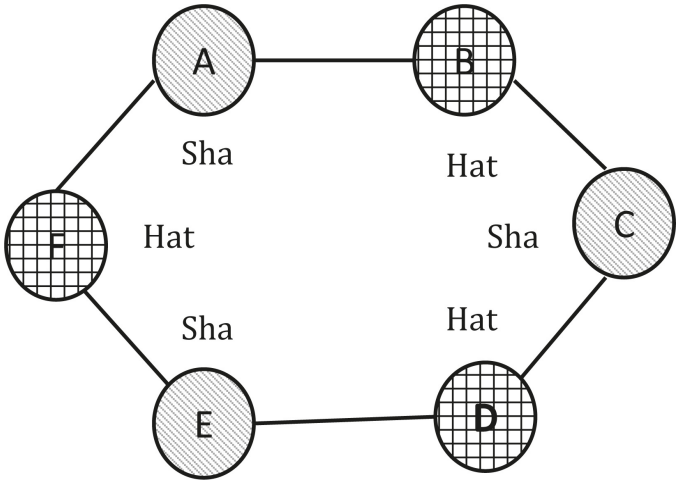
The partition we found looks like this.



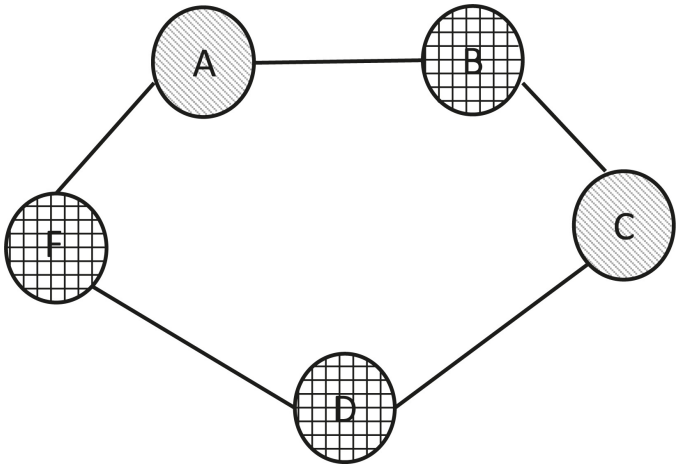
This is simply a redrawing of your social circle graph. It turns out the graph above has a name—it is a *bipartite graph*. A bipartite graph is a graph whose vertices can be divided into two independent sets, U and V, such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. We can also say that no edge connects vertices of the same set. In the above example, $U = \{A, C, E, F\}$ and $V = \{B, D, G, H, I\}$.

A graph is bipartite if it can be colored using two colors, such that no two vertices that are adjacent (i.e., share an edge) have the same color. This is, of course, a restatement of our original problem, with colors substituted for dinner nights. We will sometimes refer to the coloring constraint as the *adjacency* constraint.

You might think that a bipartite graph can't have cycles. It is possible to color a graph with cycles comprising an *even* number of vertices using two colors: Shaded and Hatched. For example, see the graph below.



$U = \{A, C, E\}$ and $V = \{B, D, F\}$, and we can invite members of U on one day, and members of V on the second. However, it is not possible to color a graph with two colors if it has a cycle involving an *odd* number of vertices. Consider F, G, and H in our second example, which has nine vertices. There, G and H have an edge between them. F, G, and H are in a 3-cycle, so that graph is not bipartite. The 5-cycle in the graph below renders it non-bipartite.



Changing the color of A to Hatched is not going to help. B and F need to be Shaded, and then C and D need to be Hatched, which violates an adjacency constraint.

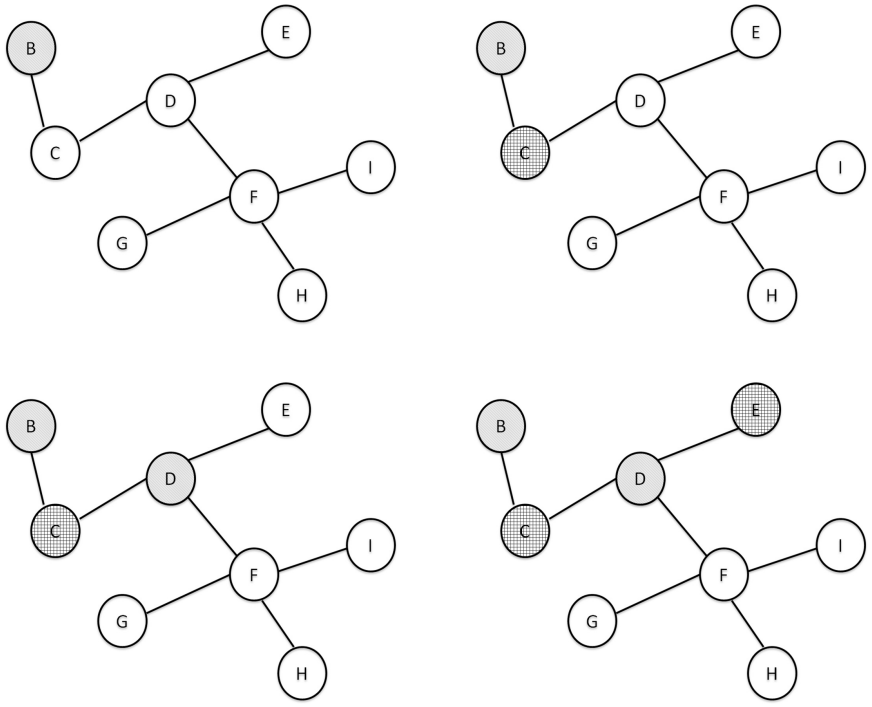
Checking If a Graph Is Bipartite

We need an algorithm that successfully colors a graph's vertices with two colors, obeying the adjacency constraint if the graph is bipartite, or telling us that the graph is not bipartite. One color will correspond to the set U, and the other to the

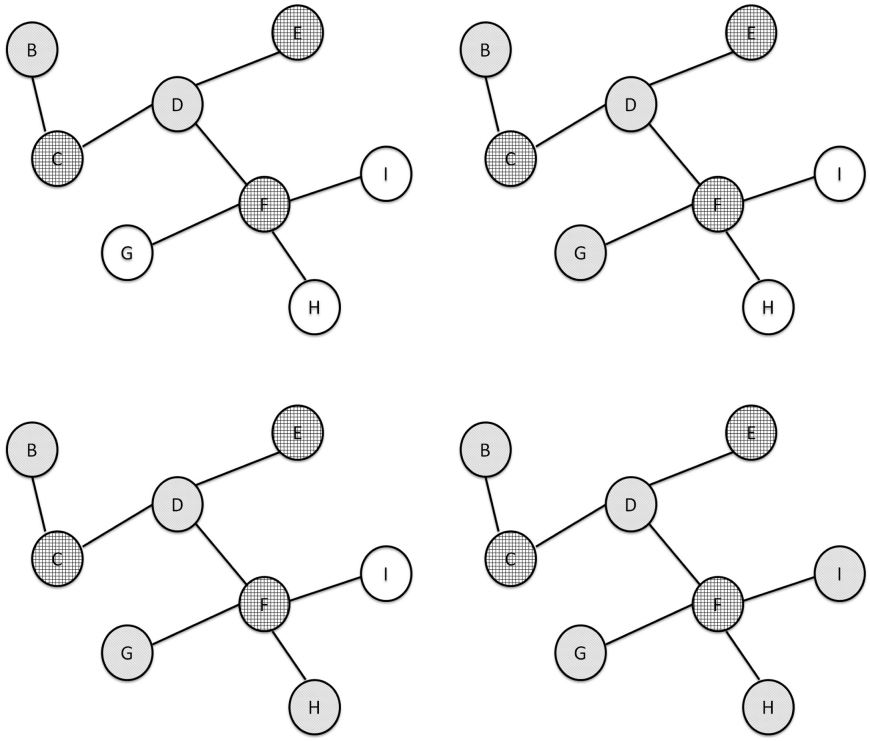
set *V*. Here's a possible algorithm that uses a technique called *depth-first search*.

. *color* = Shaded, vertex = starting vertex *w*.
. If *w* has not yet been colored, color vertex *w* with *color*.
. If *w* has already been colored with a different color than *color*, the graph is not bipartite. Return **False**.
. If *w* has already been colored with the correct color, return **True** and the (unmodified) coloring.
. Flip *color*: Shaded to Hatched, or Hatched to Shaded.
. For each neighbor *v* of *w*, recursively call the procedure with *v* and *color* (i.e., go to step 2 with *w* = *v*). If any of the recursive calls return **False**, then return **False**.
. The graph is bipartite. Return **True** and return the coloring.

Let's run this algorithm on an example graph (shown below), beginning with coloring vertex *B* Shaded. *C* is the only vertex connected to *B* and is colored next. From *C* we go to *D*, since *B* is already colored. Once we color *D*, since *C* is already colored, we have the choice of coloring *E* or *F*, and we color *E* first.



Since *E* has no neighbors other than *D*, we next move to *F*, which is a neighbor of *D*. We color the neighbors of *F* in the order *G*, *H*, *I* (see the following graph).



You may have noticed that our example above omitted vertex/friend *A*, which was in the social circle we presented earlier. This is because *A* is not connected to any of the other vertices—it does not have any neighbors. Of course, *A* could be colored either Shaded or Hatched.

We'll assume our input graph is such that we can reach all vertices from a specified start vertex. One exercise at the end of the puzzle chapter will deal with the general case of possibly disconnected components in the input graph.

Graph Representation

Before we dive into coding the algorithm, we have to choose a data structure for our graph. The data structure should allow us to perform the operations we need for the algorithm: Take a vertex, find its neighbors, then the neighbors' neighbors, and so on. Here's a graph representation, based on Python dictionaries, that will serve our needs. It represents the graph we just looked at.

```
graph = {'B': ['C'],
         'C': ['B', 'D'],
         'D': ['C', 'E', 'F'],
         'E': ['D'],
         'F': ['D', 'G', 'H', 'I'],
         'G': ['F'],
         'H': ['F'],
```

```
    'I': ['F']}
```

The dictionary represents graph vertices and edges. We use strings to represent graph vertices: *B* in the graph picture is `'B'`, and so on. Each of the vertices is a key in the dictionary `graph`. Each line corresponds to a key-value pair, where the value is a list of edges from the vertex key. The edge is simply represented by the destination vertex of the edge. In our example, *B* has a single edge to *C*, and thus we have a single-item list for the value of the `'B'` key. On the other hand, vertex key `'F'` has a four-item list corresponding to its four edges.

We have undirected graphs in this puzzle, meaning that each edge is undirected. This means that if we can traverse an edge from vertex *B* to vertex *C*, we can traverse the edge in the opposite direction, from vertex *C* to vertex *B*. For our dictionary representation, this means that if a vertex key *X* has a vertex *Y* in its value list, then vertex key *Y* will also have vertex *X* in its value list. Check the dictionary `graph` above for this symmetry condition.

You have seen dictionaries in our anagram problem of puzzle 17 and the coin row problem of puzzle 18. They are essentially lists with more general indices that can be strings, as they are in `graph`. Here, we will use a few other operations on dictionaries that show you how powerful dictionaries are.

It is important to note that we do not want to depend on any particular order of the keys in the dictionary. The code we write should discover that the following graph representation `graph2` corresponds to a bipartite graph, and color it properly. If we start with the same vertex and color for input `graph` and `graph2`, we should get the exact same coloring.

```
graph2 = {'F': ['D', 'G', 'H', 'I'],
        'B': ['C'],
        'D': ['C', 'E', 'F'],
        'E': ['D'],
        'H': ['F'],
        'C': ['B', 'D'],
        'G': ['F'],
        'I': ['F']}
```

The execution of the algorithm shown in the eight diagrams above corresponds to either dictionary `graph` or `graph2`. The order in which vertices are colored does depend on the ordering of the value lists. This is why, for example, after vertex *D* is colored, *G* is colored before *H*, which is colored before *I*.

The code for bipartite graph coloring, below, closely follows the pseudocode presented earlier.

```
1.    def bipartiteGraphColor(graph, start, coloring, color):
2.        if not start in graph:
3.            return False, {}
4.        if not start in coloring:
5.            coloring[start] = color
6.        elif coloring[start] != color:
7.            return False, {}
8.        else:
9.            return True, coloring
10.       if color == 'Sha':
11.           newcolor = 'Hat'
12.       else:
13.           newcolor = 'Sha'
14.       for vertex in graph[start]:
15.           val, coloring = bipartiteGraphColor(graph,\
15a.                                   vertex, coloring, newcolor)
16.           if val == False:
17.               return False, {}
18.       return True, coloring
```

The arguments to the procedure are the input graph (represented as a dictionary), the vertex start that we will color first, another dictionary coloring that stores the mapping from vertices to colors, and finally, the color that we will apply to the start vertex (color).

On line 2 we check that the dictionary graph has the vertex key start in it, and if not, we return **False**. Note that it is possible that during the recursive search, we encounter a vertex 'Z' in the neighbor list of a vertex, but 'Z' does not exist in the graph representation as a key. Here's a simple example:

```
dangling = {'A': ['B', 'C'],
            'B': ['A', 'Z'],
            'C': ['A']}
```

Line 2 checks for cases like the one above.

Lines 4–9 correspond to steps 2–4 of the algorithm. If this is the first time we are encountering this vertex start, the dictionary coloring will not have the vertex in it. In this case, we color the vertex with color and add it to the dictionary (step 2). If the dictionary already has start in it, then we have to compare the existing color of the vertex with the color we would like to give it. If the colors are different, the graph is not bipartite and we return **False** (step 3). Otherwise, thus far the graph is bipartite (we might discover it is not later). We therefore return, for this

call, **True** and the current coloring (step 4).

Lines 10–13 flip the color in preparation for the recursive calls. We represent Shaded as 'Sha' and Hatched as 'Hat'. On line 14, we iterate through the vertices in graph[start], which returns the list of neighbors of start. Line 15 makes the recursive calls for each neighbor vertex, updated coloring, and the flipped color. If any of the calls returns **False**, the graph is not bipartite, and we return **False**. If all the recursive calls return **True**, we return **True** and the coloring.

If we run:

```
bipartiteGraphColor(graph, 'B', {}, 'Sha')
```

This corresponds to the start vertex *B*, an empty coloring, and the color Shaded chosen for vertex *B*. It returns:

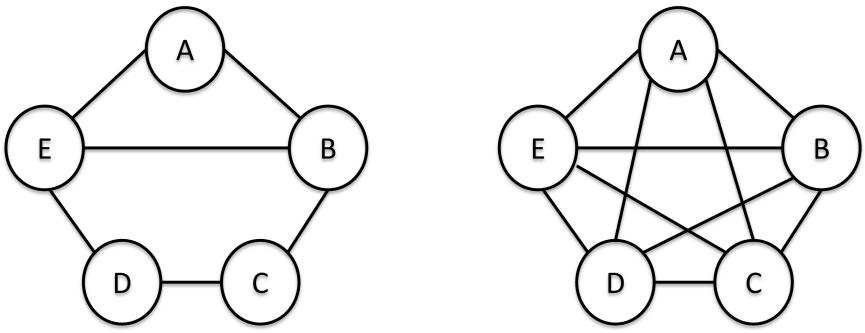
```
(True, {'C': 'Hat', 'B': 'Sha', 'E': 'Hat', 'D': 'Sha', 'H': 'Sha', 'I': 'Sha', 'G': 'Sha', 'F': 'Hat'})
```

The coloring is returned as a dictionary, and the order of keys in the dictionary is computer-platform-specific and run-specific, meaning that it may change if you run the program again with the same input. Even though vertex *B* is colored first, the key associated with vertex *C* appears first. Dictionaries make no guarantees that keys that are inserted first appear first when you print the dictionary, or when you generate all the keys using dictname.keys() for dictionary dictname. For example, our dictionary graph has 'B' listed first in the representation. If we print graph.keys(), which returns a list, we could get ['C', 'B', 'E', 'D', 'G', 'F', 'I', 'H']. Similarly, graph.values() generates all the values in the dictionary, which could produce [['B', 'D'], ['C'], ['D'], ['C', 'E', 'F'], ['F'], ['D', 'G', 'H', 'I'], ['F'], ['F']].

Graph Coloring

A coloring of a graph using at most *k* colors is called a (proper) *k*-coloring. While we have shown an efficient way of checking whether a graph can be colored using two colors, the same problem for three colors is a hard problem. That is, any known algorithm that can do this (correctly determine whether an arbitrary graph can be colored using at most three colors) could require, for some graphs, a number of operations that grows exponentially in the number of vertices in the graph.

The first results about graph coloring dealt almost exclusively with a special class of graphs called planar graphs, which arise from the coloring of maps. A planar graph is a graph that can be drawn so that no edges cross each other. Below is an example of a planar and a non-planar graph.



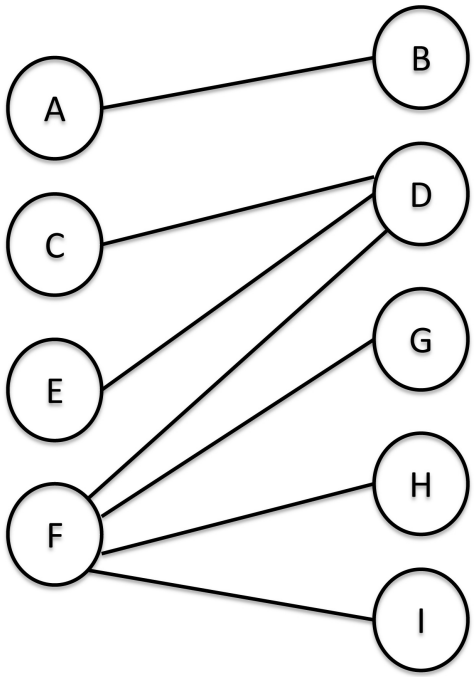
Planar Graph Non-Planar Graph

While trying to color a map of the counties of England, Francis Guthrie conjectured that four colors were sufficient to color any map so that no regions sharing a common border received the same color. This is famously called the four-color conjecture for planar graphs.

In 1879, Alfred Kempe published a paper that claimed to establish the result. In 1890, Percy John Heawood pointed out that Kempe's argument was wrong and also proved the five-color theorem using the ideas of Kempe. The five-color theorem shows that every planar map can be colored with no more than five colors. After many attempts, the four-color theorem for planar graphs was finally proved in 1976 by Kenneth Appel and Wolfgang Haken using techniques very different from Kempe's. The proof of the four-color theorem is also noteworthy for being the first major computer-aided proof.

Exercises

Exercise 1: The bipartite graph checker assumes that the graph is connected. Your social circle may contain disconnected components, as in the following example—a variant of the earlier example.



Modify the code so it works on the above type of graph. The current code will only color two vertices *A* and *B* in the graph if the start vertex argument to bipartiteGraphColor is *A* or *B*, and will leave *C* through *I* uncolored.

Create a parent procedure that invokes bipartiteGraphColor on the input graph with some starting vertex. Then check that all vertices in the input graph are colored. If not, begin with an uncolored vertex and run bipartiteGraphColor starting with this vertex. Continue till all vertices in the input graph are colored. Once you have colored all the vertices in each of the components, output dinner invitations, provided all the components are bipartite.

Exercise 2: Modify the procedure bipartiteGraphColor so it prints a cyclic path from the start vertex that cannot be colored using two colors, if such a path exists. Such a path will exist if the graph is not bipartite, and will not otherwise. The cycle may not include the start vertex, but will be reachable from the start vertex in the case where the graph is not bipartite. Suppose you are given the graph below:

```
graphc = {'A': ['B', 'D' 'C'],
          'B': ['C', 'A', 'B'],
          'C': ['D', 'B', 'A'],
          'D': ['A', 'C', 'B']}
```

Your modified procedure should output:

```
Here is a cyclic path that cannot be colored ['A', 'B', 'C', 'D', 'B']
```

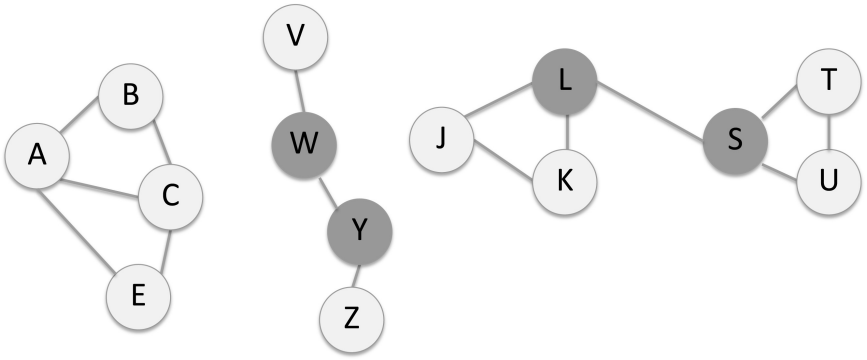
(False, {})

Exercise 3: The procedure bipartiteGraphColor embodies depth-first search. The code below makes recursive calls following a sequence of vertices.

```
14.         for vertex in graph[start]:
15.             val, coloring = bipartiteGraphColor(graph,\
15a.                                 vertex, coloring, newcolor)
```

Rather than coloring the graph, suppose we wish to find paths between pairs of vertices. Write a function findPath that finds and returns a path from a start vertex to an end vertex if such a path exists, and returns **None** if it does not. If we run findPath on the dictionary graph, with start vertex 'B' and end vertex 'I', it should find the path ['B', 'C', 'D', 'F', 'I'].

Puzzle Exercise 4: A vertex in an undirected connected graph is an *articulation point* if removing it (and edges through it) disconnects the graph. Articulation points are useful for designing reliable networks because they represent vulnerabilities in a connected network—single points whose failure would split the network into two or more disconnected components. Below are graphs whose articulation points are shaded.



Design and code an algorithm that will determine whether a given graph has an articulation point or points, and report all such vertices if they exist.

20

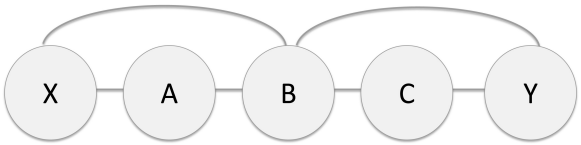
Six Degrees of Separation

Okay, I was in a movie with an extra, Eunice, whose hairdresser, Wayne, attended Sunday school with Father O'Neill, who plays racquetball with Dr. Sanjay, who recently removed the appendix of Kim, who dumped you sophomore year. So you see, we're practically brothers.
—Kevin Bacon in a Visa Check Card commercial.

> Programming constructs and algorithmic paradigms covered in this puzzle: Set operations. Breadth-first graph traversal using sets.

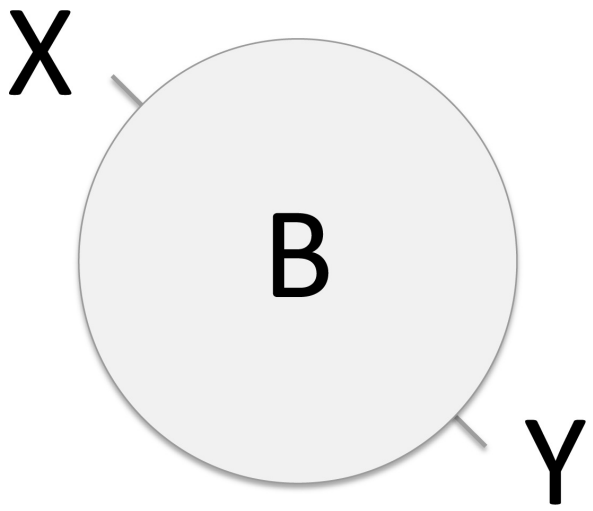
Six degrees of separation is the theory that everyone is six or fewer steps away, by way of introduction, from any other person in the world, so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps. Each step is a degree of separation. With a loose definition of friend, one can usually make the case for two random people, and so six degrees of separation has become a popular theory.

To determine the degree of separation between two people, we need to find the *shortest* relationship between them. *A* might have a best friend *B*, and also a friend *C* who is friends with *B*. The relationship through *C* between *A* and *B* is of length 2, but the direct relationship gives the degree of separation between *A* and *B* as 1. Similarly, in the graph below, where the vertices represent people and the edges represent relationships, the degree of separation between *X* and *Y* is 2, not 3 or 4. The degree of separation between *A* and *C* is also 2.



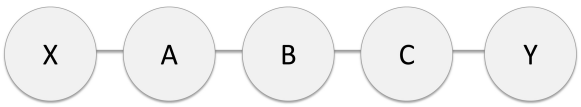
The degree of separation for a graph is the maximum degree of separation between any pair of vertices. For the graph above, the degree of separation is 2. Every vertex is reachable from every other vertex in 2 steps.

It is important to understand the difference between the degree of separation between two vertices in a graph, and the degree of separation of the graph. The latter is also called the diameter of the graph. Understand that even if everyone in the universe is reachable from a particular person in *k* or fewer steps, it does *not* mean that the degree of separation in the universe is *k*. It is at least *k*, but could be significantly higher. The intuitive way of seeing this is illustrated below.



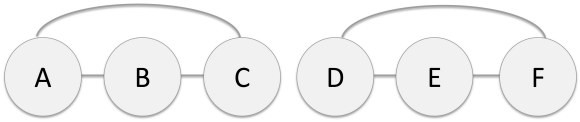
The circle represents the universe. The center *B* of the circle is only, at most, the radius away from any point on the circle or within it. But of course, two points on the circle that are diametrically opposite (e.g., *X* and *Y*) are a diameter apart, which is twice the radius.

In the more concrete example below, the degree of separation from *B* (the center of the graph) to any vertex is at most 2. But *X* and *Y* are 4 steps from each other, so the degree of separation of the graph is 4.



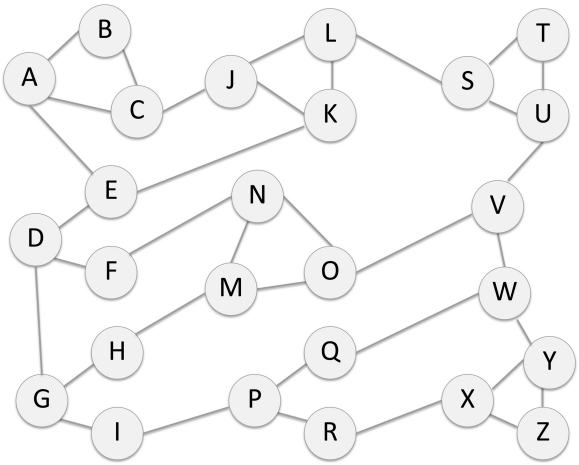
Exercise: Can you argue that given a vertex *T* in a graph, if the degree of separation between *T* and any other vertex is at most *k*, then the degree of the separation of the graph is *at most* 2*k*?

We will assume in this puzzle that our graph is connected, that is, every vertex can be reached from every other vertex. If we have two clusters of vertices, as shown below, the degree of separation for each of the clusters is 1, but the degree of separation for the graph comprising both clusters is infinity, since *A* cannot reach *F*.



With the preliminaries out of the way, let's raise the stakes with some bigger examples.

Does the graph below violate the six-degrees-of-separation hypothesis? What is the maximum degree of separation in this graph between any pair of vertices? How would you compute it?



Breadth-First Search

Our algorithm will work as follows: We'll start from a particular vertex *S* (for source) and find the shortest path from *S* to every other vertex in the graph. This will determine the degree of separation between *S* and every other vertex. If the degree of separation between *S* and any other vertex is k_S, then we immediately know that the degree of separation of the graph *d* is such that $k_S \leq d \leq 2k_S$. But how will we determine the exact *d*?

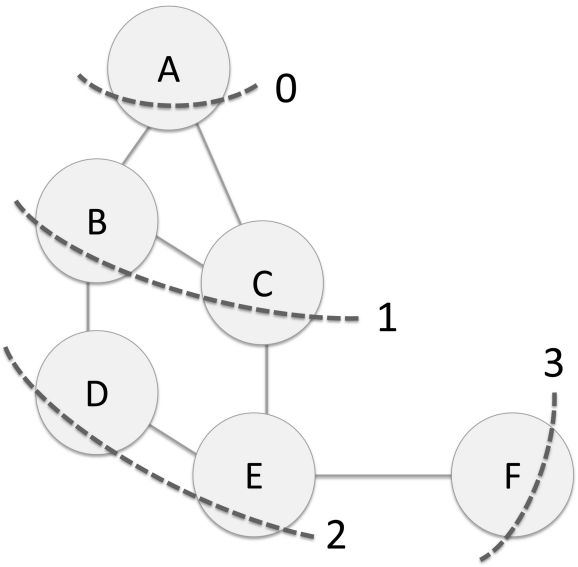
If we simply start with every vertex as a source and run the shortest-path algorithm from each source, then we can pick the maximum k_S that we see, because we have computed the degree of separation between each pair of vertices in the graph.

What we now need is an algorithm to determine, given a source vertex, the *shortest* path to every other vertex, where the path is a sequence of edges. The number of edges is the length of the path—what we called the number of steps required to reach that vertex. We did look at a path-finding algorithm in puzzle 19, but that algorithm only guaranteed that it would reach every connected vertex in the graph from a source vertex, and made no guarantees that the sequence of edges traversed in reaching any particular vertex would be minimum.

Breadth-first search (rather than the depth-first search algorithm of puzzle 19) meets our needs. Breadth-first search, as its name indicates, collects all vertices reachable from the source vertex in one step (i.e., one-edge traversal). These vertices form the new frontier of the search, replacing the source vertex, which is the initial frontier. Given a frontier, we find a new collection of vertices—those that are reachable within one step from *any* of the frontier vertices. Crucially, we don't bother with collecting vertices that have already been visited—for example, the source vertex. We only include a vertex in any frontier exactly once.

The first frontier, the source, is trivially reachable in 0 steps from the source. The second frontier comprises vertices that are reachable in one step; the third comprises vertices reachable in two steps, and so on. When we have visited all the vertices, the search ends. The vertices in the last frontier are the ones that have the maximum degree of separation from the source vertex *S*, and are only reachable in, say, k_S steps.

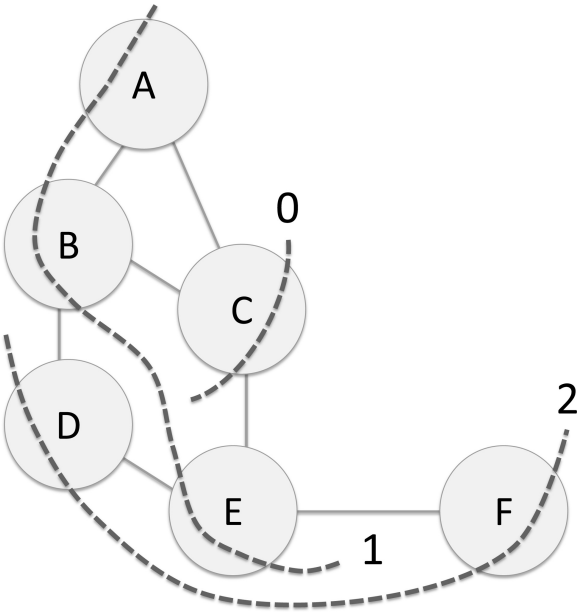
Here's how the algorithm works on the graph below, with source vertex *A*.



We have drawn the frontier as a line through the vertices belonging to each frontier and numbered the frontiers in the order they are generated. Each vertex belongs to exactly one frontier. Since *B* and *C* are in frontier 1, we look for vertices not yet visited that are reachable from *B* and *C* in one step. And this yields frontier 2: *D* and *E*.

While there is a path in the graph from *A* to *F* with 5 edges, $A \rightarrow C \rightarrow B \rightarrow D \rightarrow E \rightarrow F$, the shortest path is $A \rightarrow C \rightarrow E \rightarrow F$, with 3 edges. In recursive depth-first search, it is quite possible that we first reach *F* using the longer path. Breadth-first search computes shortest paths, and the k_A computed in this execution is 3.

Let's say we started with the vertex *C* as the source vertex instead. The frontiers would instead look like this:



A, *B*, and *E* are reachable in one step, and *D* and *F* in two steps. This means that $k_C = 2$.

Sets

Each frontier in our breadth-first search algorithm is a set of vertices, mathematically speaking. That is, the order of vertices does not matter, and vertices are not repeated. Sets in Python allow us to represent mathematical sets (i.e., unordered collections of objects without repeats), and Python provides operations to manipulate them. We used sets in discovering implications in the sudoku puzzle (puzzle 14).

Here's an example set:

```
frontier = {'A', 'B', 'D'}
```

We can add to a set and delete elements from the set as follows:

```
frontier.add('F')
frontier.remove('A')
print(frontier)
```

This produces:

```
{'D', 'B', 'F'}
```

Note that a `KeyError` exception is thrown if the element to be removed is not in the set. You can check if an element is or is not in a set with, for example, `'A' in frontier`, and `'A' not in frontier`, respectively.

You can create an empty set and add to it, as shown in the following:

```
frontier = set()
frontier.add('A')
```

Python supports set operations such as intersection, union, and difference.

Using Sets in Breadth-First Search

We will use the same graph dictionary representation we used in puzzle 19. Below is the representation for the example we just executed the algorithm on.

```
small = {'A': ['B', 'C'],
         'B': ['A', 'C', 'D'],
         'C': ['A', 'B', 'E'],
         'D': ['B', 'E'],
         'E': ['C', 'D', 'F'],
         'F': ['E']}
```

Each vertex is a key in the dictionary. Each line above is a key-value pair, with the value being a list of vertices that are connected to the vertex key via edges. As before, we are dealing with undirected graphs, that is, graphs with undirected edges.

The code for our breadth-first search procedure is shown below.

```
1.  def degreesOfSeparation(graph, start):
2.      if start not in graph:
3.          return -1
4.      visited = set()
5.      frontier = set()
6.      degrees = 0
7.      visited.add(start)
8.      frontier.add(start)
9.      while len(frontier) > 0:
10.         print (frontier, ':', degrees)
11.         degrees += 1
12.         newfront = set()
13.         for g in frontier:
14.             for next in graph[g]:
15.                 if next not in visited:
16.                     visited.add(next)
17.                     newfront.add(next)
18.         frontier = newfront
```

```
19.         return degrees - 1
```

The procedure just takes the graph and a start vertex as arguments. It computes the shortest path to every other vertex in the graph. If the start vertex is not in the graph dictionary, the procedure aborts and returns `-1` (lines 2−3).

The data structures required for breadth-first search correspond to Python sets. We need to keep track of what vertices have been visited, since we have to ensure that each vertex is in exactly one frontier. The set of visited vertices and the current frontier are created as empty sets (lines 4−5). The variable `degrees` numbers the frontiers and is initialized to `0` (line 6). We start our search on lines 7 and 8 by visiting the start vertex explicitly. It is added to the visited and frontier sets.

The **while** loop (lines 9−18) implements breadth-first search. While we have a nonempty frontier, we search from each vertex in the current frontier for new vertices that have not been visited yet. The outer **for** loop (lines 13−17) goes through each vertex in the current frontier. For each vertex in the frontier, we look at each neighbor in the inner **for** loop (lines 14−17). If these neighbors have not been visited yet (check on line 15), they are marked as visited (line 16) and added to the new frontier (line 17). Once we exit the outer **for** loop, we simply move to the next frontier by setting the current frontier to the new one (line 18).

After we exit the **while** loop, we have processed all the vertices and reached what we will call the final frontier (in honor of Star Trek). We therefore return the number assigned to the final frontier. This is the maximum degree of separation between the source vertex and any of the other vertices in the graph.

Let’s run the code on our small graph example.

```
degreesOfSeparation(small, 'A')
```

It produces exactly what was shown in the picture earlier.

```
{'A'} : 0
{'C', 'B'} : 1
{'E', 'D'} : 2
{'F'} : 3
```

Where’s the fun in that? Let’s run the code on the large graph.

```
large = {'A': ['B', 'C', 'E'], 'B': ['A', 'C'],
         'C': ['A', 'B', 'J'], 'D': ['E', 'F', 'G'],
         'E': ['A', 'D', 'K'], 'F': ['D', 'N'],
         'G': ['D', 'H', 'I'], 'H': ['G', 'M'],
         'I': ['G', 'P'], 'J': ['C', 'K', 'L'],
         'K': ['E', 'J', 'L'], 'L': ['J', 'K', 'S'],
```

```
                'M': ['H', 'N', 'O'], 'N': ['F', 'M', 'O'],
                'O': ['N', 'M', 'V'], 'P': ['I', 'Q', 'R'],
                'Q': ['P', 'W'], 'R': ['P', 'X'],
                'S': ['L', 'T', 'U'], 'T': ['S', 'U'],
                'U': ['S', 'T', 'V'], 'V': ['O', 'U', 'W'],
                'W': ['Q', 'V', 'Y'], 'X': ['R', 'Y', 'Z'],
                'Y': ['W', 'X', 'Z'], 'Z': ['X', 'Y']}
degreesOfSeparation(large, 'A')
```

This produces:

```
{'A'} : 0
{'C', 'B', 'E'} : 1
{'J', 'K', 'D'} : 2
{'F', 'L', 'G'} : 3
{'I', 'S', 'N', 'H'} : 4
{'T', 'M', 'O', 'U', 'P'} : 5
{'Q', 'V', 'R'} : 6
{'X', 'W'} : 7
{'Z', 'Y'} : 8
```

This tells us that the degree of separation of the graph is 8.

Wrong! It tells us that it is at least 8. We need to run it on all possible starting vertices. We won't bore you with printing out each of the results—we'll cut to the chase. As shown below, choosing start vertex *B* yields the maximum degree of separation between any pair of vertices. Running:

```
degreesOfSeparation(large, 'B')
```

Produces:

```
{'B'} : 0
{'C', 'A'} : 1
{'E', 'J'} : 2
{'K', 'D', 'L'} : 3
{'F', 'S', 'G'} : 4
{'U', 'I', 'T', 'N', 'H'} : 5
{'V', 'O', 'M', 'P'} : 6
{'Q', 'R', 'W'} : 7
{'X', 'Y'} : 8
{'Z'} : 9
```

The shortest way of getting from vertex *B* to vertex *Z* (or vice versa) traverses 9 edges.

As you run `degreesOfSeparation` for different starting vertices, you will discover that the "center" of the graph is vertex *U*. Running:

```
degreesOfSeparation(large, 'U')
```

Produces:

```
{'U'} : 0
{'T', 'S', 'V'} : 1
{'O', 'W', 'L'} : 2
{'J', 'M', 'K', 'Y', 'Q', 'N'} : 3
{'F', 'P', 'E', 'C', 'H', 'Z', 'X'} : 4
{'A', 'B', 'D', 'I', 'R', 'G'} : 5
```

k_U = 5 for the large graph, and it is the smallest value among all the start vertices. We got lots of interesting information about the graph by writing about twenty lines of code. We hope you are convinced by now that programming is useful and computer science is cool. If you aren't convinced, there is one more puzzle that might do the trick.

History

The theory of six degrees of separation was first proposed in 1929 by Frigyes Karinthy, a Hungarian writer, in a short story called "Chains." Ithiel de Sola Pool (MIT) and Manfred Kochen (IBM) set out to prove the theory mathematically in the 1950s, and made progress in formulation, but did not come up with a satisfactory proof.

In 1967, American sociologist Stanley Milgram referred to the theory as "the small-world problem" and devised an experimental way to test it. Randomly selected people in the Midwest were asked to send packages to a stranger in Massachusetts. The stranger's name and occupation were known to the senders, but their specific location was unknown. The instructions to the senders were as follows: "Send the package to a person you know on a first-name basis who you think is most likely to know the target personally. Give the same instructions to your friend." This was supposed to continue until the package was personally delivered to its target.

Surprisingly, it only took (on average) between five and seven intermediaries to get each package delivered. Milgram's findings were published in *Psychology Today* and the phrase "six degrees of separation" was born. His findings were criticized after it was discovered that he based his conclusion on a very small number of packages. Decades later, Duncan Watts, a professor at Columbia University, recreated Milgram's experiment on the Internet on a much larger scale. Watts used an e-mail message as the "package" to be delivered, and found that the average number of intermediaries was indeed six!

A site called Six Degrees, launched in 1997, is considered by many to be the

first social networking site. Modern sites like Facebook and Twitter have effectively lowered the number of intermediaries in the chain, arguably to almost zero.

Exercises

Exercise 1: Write a procedure that takes the graph and a pair of vertices in the graph as arguments, and determines the degree of separation between the pair of vertices. Then write another procedure that determines the degree of separation for every pair of vertices in the graph, invoking the first procedure, and returns the maximum value as the degree of separation of the graph.

Of course, another way of doing this is to run the degreesOfSeparation procedure to completion for each start vertex in the graph, find the vertex with maximum separation from each start vertex, and report the maximum value across all the runs. This is what we did to find the degree of separation of 9 for our large social circle.

Exercise 2: One of the annoying things about our graph dictionary representation for undirected graphs is that we have to represent each undirected edge between vertices *A* and *B* as two edges: one from *A* to *B*, and another from *B* to *A*. It is easy to make mistakes when writing the representations for large graphs. In fact, we made numerous mistakes in manually converting the pictorial representation of our large 26-vertex example to the graph dictionary large. Write a procedure, as we did, that checks a graph dictionary representation for symmetry—if there is an edge from any vertex *X* to a vertex *Y*, there should be an equivalent edge from *Y* to *X*.

Exercise 3: While the code shown computes and prints each frontier, it does not explicitly tell you the path between any pair of vertices. Write a procedure that, given a graph and a pair of vertices, prints the shortest path between the pair. For vertices *B* and *Z*, we should get:

$$B \rightarrow C \rightarrow J \rightarrow L \rightarrow S \rightarrow U \rightarrow V \rightarrow W \rightarrow Y \rightarrow Z$$

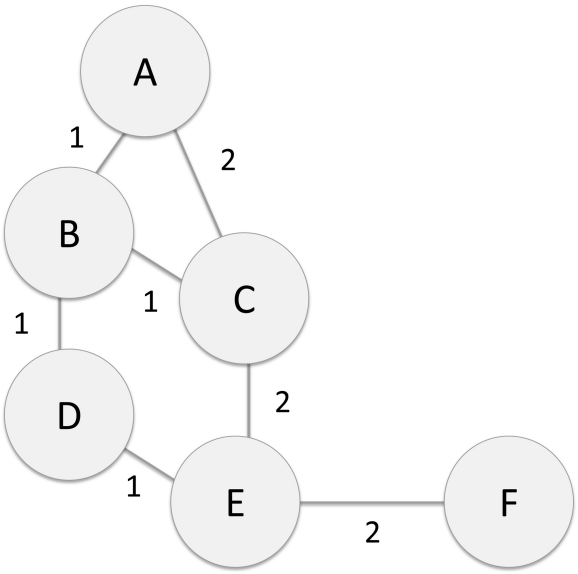
or possibly

$$B \rightarrow A \rightarrow E \rightarrow D \rightarrow G \rightarrow I \rightarrow P \rightarrow R \rightarrow X \rightarrow Z$$

Hint: You will have to store all the frontiers and work backward from the end vertex. Find a vertex *W* in the frontier immediately before reaching the end

vertex that has an edge to the end vertex—there has to be at least one such *W*. Then, find a vertex in the frontier immediately before reaching *W* that has an edge to *W*. And so on till you get to the start vertex, that is, the first frontier.

Puzzle Exercise 4: Suppose that not all edges are created equal: Some of them represent distant, as opposed to close, relationships. We will assign a weight of 2 to an edge representing a distant relationship and a weight of 1 to an edge representing a close relationship. Below is a weighted graph example, followed by its dictionary representation.




```
smallw = {'A': [('B', 1), ('C', 2)],
          'B': [('A', 1), ('C', 1), ('D', 1)],
          'C': [('A', 2), ('B', 1), ('E', 2)],
          'D': [('B', 1), ('E', 1)],
          'E': [('C', 2), ('D', 1), ('F', 2)],
          'F': [('E', 2)]}
```

Each element in the value list for a vertex key is a 2-tuple that contains the destination vertex and the weight of the edge. We are dealing with undirected edges, and the weight of the edge in either direction can be assumed to be the same.

We'll define the *weighted degree of separation* between two vertices as the minimum-weight path between the vertices, where the weight of a path is the sum of weights of constituent edges. The weighted degree of separation from vertex *A* to vertex *C* is 2, since the direct edge from *A* to *C* is a distant edge. Similarly, the weighted degree of separation from *A* to *F* is 5, through the path *A*

→ *B* → *D* → *E* → *F* of weight 1 + 1 + 1 + 2 = 5. The path with fewer edges, *A* → *C* → *E* → *F*, has greater weight 2 + 2 + 2 = 6.

Write a procedure `weightDegreesOfSeparation` that computes the *maximum* weighted degree of separation from a start vertex to any other vertex. This will require you to compute the weighted degree of separation from the start vertex to every other vertex.

Hint: Think graph transformation as opposed to modifying breadth-first search. Make sure your transformed graph—represented as a dictionary—satisfies the symmetry relationship, by running it on your code from exercise 2.

21

Questions Have a Price

If you have to ask how much it costs, you can't afford it.
—J. P. Morgan

> Programming constructs and algorithmic paradigms covered in this puzzle: Object-oriented programming. Binary search trees.

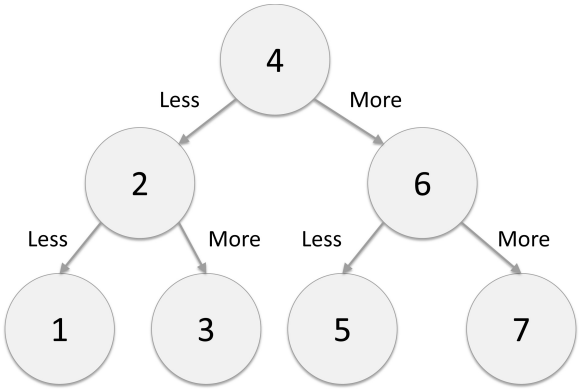
We've all played the twenty questions game. Here's one with a twist.

Your friend is going to think of a number from 1 to 7. Your job is to guess the number using the fewest number of guesses. Once you have guessed it correctly, it is your turn to think of a number and your friend's turn to guess. The two of you have nothing better to do than play this game umpteen times, keeping track of all the guesses made by each person for each round, where in each round both of you have a chance to guess. Afterward, you'll go to dinner, and the person with more guesses pays. So you are very motivated to win.

Now some details of how the game is played. We'll call the person who thought of the number Thinker and the person who guesses Guesser. Thinker has to write the number down before Guesser starts guessing to ensure that Thinker does not cheat and change the number on the fly. When Guesser guesses a number, Thinker responds with Correct, Less, or More. These responses have the obvious meanings.

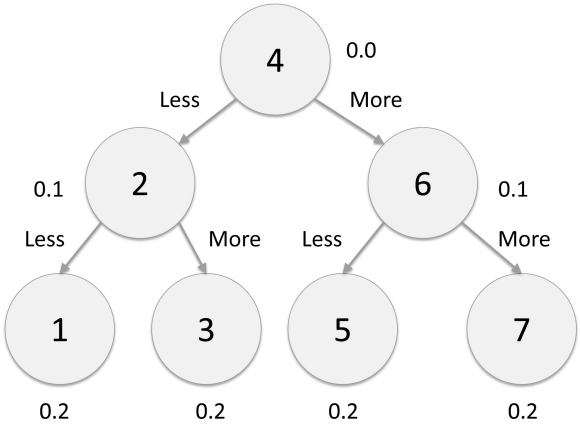
At this point, you are probably thinking binary search is going to be the best strategy. With numbers between 1 and 7 inclusive, you are going to need at most

three guesses to find the number, since $\log_2 7$ is greater than 2 but less than 3. Here's what the binary search tree (BST) looks like:



The BST above represents an execution of binary search. The root is the first guess, 4. You move to 2 with a Less response, move to 6 with a More response, and celebrate with a Correct response. If your friend thought 4, you use only one guess, and if your friend guessed 7, you will need three guesses: 4, 6, 7. The BST is a cool data structure and satisfies this invariant: All vertices to the left of a vertex will have numbers less than the vertex's number, and all vertices to the right of a vertex will have numbers larger than the vertex's number. This is true of every vertex in a BST—a recursive property. A vertex may have one, two, or no children.

Your friend knows about binary search and BSTs, so you figure it is going to be a toss-up who pays for dinner. But then you realize that your friend is constantly picking odd numbers because the odd numbers are at the bottom of the BST, and you need more guesses for these. In fact, you estimate that the probability of a number being chosen by your friend in each round is exactly as given below (and now listed on the BST figure below):



Pr(1) = 0.2 Pr(2) = 0.1 Pr(3) = 0.2 Pr(4) = 0
Pr(5) = 0.2 Pr(6) = 0.1 Pr(7) = 0.2

You reason that you can win this game by choosing a different BST, which needs fewer guesses for the odd numbers. You know your friend is stubborn and isn't going to change the probabilities, so if you come up with a BST that requires, on average, fewer guesses, you are going to win.

Can you synthesize a different BST that minimizes the expected number of guesses you need to make, given the probabilities above? The following quantity should be minimized:

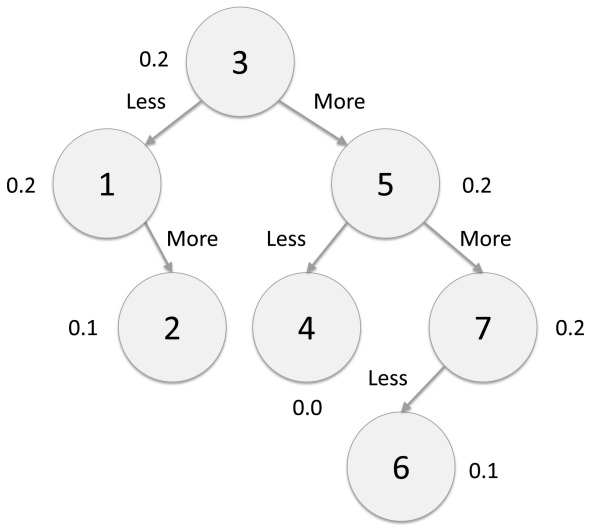
$$weight = \sum_{i=1}^{7} Pr(i) \cdot (D(i)+1)$$

D(*i*) is the depth of the number *i* in your BST. For the conventional BST shown earlier, the above quantity is 0.2 · 3 + 0.1 · 2 + 0.2 · 3 + 0 · 1 + 0.2 · 3 + 0.1 · 2 + 0.2 · 3 = 2.8. For *i* = 1, the number 1 has probability 0.2 at depth 3, and for *i* = 2, the number 2 has probability 0.1 at depth 2. And so on.

You can do appreciably better with a different BST. A free dinner beckons.

Here's the optimal BST for the given probabilities that minimizes the quantity:

$\Sigma_i \, Pr(i) \cdot (D(i)+1).$



The first thing to note is that it has vertices at greater depth, 4, than the "balanced" BST, which has a maximum depth of 3 and an average number of

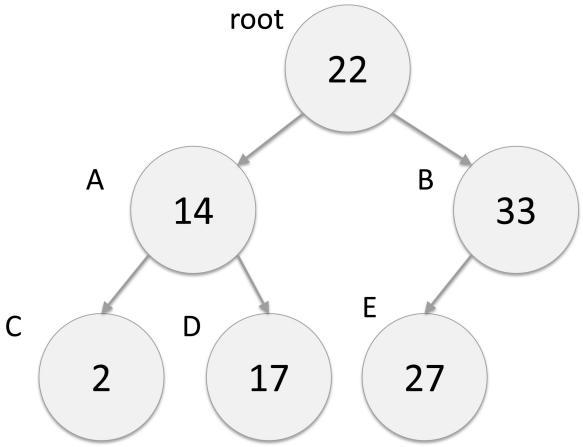
guesses of 2.8. The average number of guesses for the above BST, which is what we are calling the *weight* of the BST, is 0.2 · 2 + 0.1 · 3 + 0.2 · 1 + 0 · 3 + 0.2 · 2 + 0.1 · 4 + 0.2 · 3 = 2.3 < 2.8. This is the best you can do for the given probabilities.

Perhaps you came up with this BST through trial and error. Of course, the next time you play your friend or someone else, you might have to synthesize a different BST that optimizes a different set of probabilities. Not surprisingly, we want to write a program that automatically produces the optimal BST with minimum weight, given a vector of probabilities.

Binary Search Trees Using Dictionaries

We have already seen how we can represent graphs using dictionaries. A BST is a special kind of graph, so we can clearly use a dictionary representation.

Here's an example BST.



And here's what the representation using a dictionary looks like.

```
BST = {'root': [22, 'A', 'B'],
               'A': [14, 'C', 'D'],
               'B': [33, 'E', ''],
               'C': [2, '', ''],
               'D': [17, '', ''],
               'E': [27, '', '']}
```

'root' corresponds to the root vertex, with number 22, and BST['root'] gives us a value list (of length 3) that has the number associated with the root vertex, followed by the left child and then the right child. Therefore, BST['root'][1] gives us the left child, which is vertex 'A'. BST[BST['root'][1]] would give us the value list

corresponding to vertex 'A', namely, [14, 'C', 'D'].

The empty string '' represents a child that does not exist. For example, the right child of the vertex named 'B' with number 33 does not exist. Leaf vertices, such as vertex 'E' with number 27, have no children.

Notice that our earlier BST picture examples did not have names for vertices. However, our BST dictionary representation does. We'll assume that all numbers in a BST are unique, so numbers can uniquely identify vertices. It would be nice if we could simplify our representation and use the numbers directly as keys in our dictionary representation. We could try to do this:

```
BSTnoname = {22: [14, 33],
             14:  [2, 17],
             33:  [27, None],
              2:   [None, None],
             17:  [None, None],
             27: [None, None]}
```

However, we have a problem here. How do we know which number corresponds to the root? Remember, we cannot depend on the order of keys in a dictionary. If we enumerate the keys in a dictionary and print them, we might get the order shown above or a different order with, say, 33 being the first key printed, which is not the root of our BST.

Of course, we could discover the root by going through the whole BST and determining that 22 does not appear as a number in any of the value lists in the dictionary—therefore, it has to be the root. But this involves significant computation that grows with the number of vertices—call it *n*—in the BST. Usually, we use BSTs so we can perform operations such as looking up a number using $c \log n$ operations, where c is a small constant. We have to have a way of marking the root. We could do this outside the dictionary structure using a list:

```
BSTwithroot = [22, BSTnoname]
```

Pretty cumbersome! So we will stick to the original representation and show you how we can create a BST, look up numbers in the BST, and more. And then we will dump the dictionary representation for an object-oriented programming representation!

Operations on BSTs under a Dictionary Representation

Let's take a look at how we can use the dictionary representation for BSTs.

We'll look at code that checks if a number exists in the BST, code to insert a new number into the BST, and finally, code that produces all the numbers in the BST in sorted order. All these procedures will traverse the BST recursively from the root to the leaves. A leaf in the BST is a vertex without children. For instance, in our dictionary example, 2, 17, and 27 are leaves.

Here's code to look up a number in a BST.

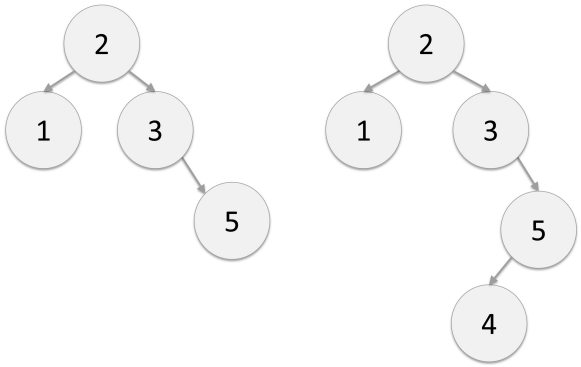
```
1.  def lookup(bst, cVal):
2.      return lookupHelper(bst, cVal, 'root')

3.  def lookupHelpe (bst, cVal, current):
4.      if current == '':
5.          return False
6.      elif bst[current][0] == cVal:
7.              return True
8.      elif (cVal < bst[current][0]):
9.          return lookupHelper(bst, cVal, bst[current][1])
10.     else:
11.         return lookupHelper(bst, cVal, bst[current][2])
```

We look up a number starting with the root vertex, which we identify with the name 'root' (line 2). If the current vertex we are looking at is empty (represented by the empty string ''), we return **False** (lines 4–5). If the current vertex's value is equal to the value we are searching for, we return **True** (lines 6–7). Otherwise, if the value we are searching for is less than the current vertex's value, we recursively search the left child of the current vertex (lines 8–9). The final case is that the value we are searching for is greater than the current vertex's value, in which case we recursively search the right child of the current vertex (lines 10–11).

Next, let's look at how we can modify the BST by inserting a new number into it. We have to insert it in such a way that the BST property is maintained. We pretend we are looking for this number in the BST and walk down from the root, going left or right based on the values encountered. When we reach a leaf and haven't found the number, depending on whether the number is smaller or larger than the leaf, we create a left or right child of the leaf vertex, respectively, for the number to be inserted.

Suppose we have the BST shown to the left (in the diagram below) and we wish to insert 4 into it. The result is shown on the right.



We go right at 2 (since 4 > 2) to 3, right at 3 to 5 (4 > 3), and left at 5 (4 < 5) to create a new vertex corresponding to 4.

The insert algorithm code shown below assumes that the number (called val) does not exist in the BST—you can imagine that it is only called for the number if lookup produces **False**. It also assumes that there is already a vertex called 'root'. The root needs to have a number associated with it and initially has no children. Children are added using insert. Note also that unlike in our pictorial example above, we have to deal with the vertex names as well as vertex values in the code.

```
1.  def insert(name, val, bst):
2.      return insertHelper(name, val, 'root', bst)

3.  def insertHelper(name, val, pred, bst):
4.      predLeft = bst[pred][1]
5.      predRight = bst[pred][2]
6.      if ((predRight == '') and (predLeft == '')):
7.          if val < bst[pred][0]:
8.              bst[pred][1] = name
9.          else:
10.             bst[pred][2] = name
11.         bst[name] = [val, '', '']
12.         return bst
13.     elif (val < bst[pred][0]):
14.         if predLeft == '':
15.             bst[pred][1] = name
16.             bst[name] = [val, '', '']
17.             return bst
18.         else:
19.             return insertHelper(name, val, bst[pred][1], bst)
20.     else:
```

```
21.             if predRight == '':
22.                 bst[pred][2] = name
23.                 bst[name] = [val, '', '']
24.                 return bst
25.             else:
26.                 return insertHelper(name, val, bst[pred][2], bst)
```

The procedure insert simply calls insertHelper, with the root vertex assumed to have the name 'root'. Otherwise, we would have to store the name of the root somewhere and use it to begin the search.

In insertHelper we have several base cases strewn around the code. If we have a vertex with no children (the check is on line 6), we can insert the number as the left or right child of the current vertex (lines 7–12) and we are done. Otherwise, if the number to be inserted is less than the number of the vertex (line 13), we can insert the number as the left child, provided the left child does not already exist (lines 15–17), and stop. If the left child exists, we recursively call insertHelper with the left child. Lines 20–26 handle the case where the number to be inserted is greater than the number of the vertex, and mimic lines 13–19.

To create the BST corresponding to BST given earlier, we unfortunately can't just use insert on an empty dictionary. We have to create an empty tree by creating an empty dictionary, then add the root vertex explicitly, and then add vertices with names and numbers as shown below:

```
BST = {}
BST['root'] = [22, '', '']
insert('A', 14, BST)
insert('B', 33, BST)
insert('C', 2, BST)
insert('D', 17, BST)
insert('E', 27, BST)
```

The last insert returns:

```
{'C': [2, '', ''], 'root': [22, 'A', 'B'], 'E': [27, '', ''],
'A': [14, 'C', 'D'], 'B': [33, 'E', ''], 'D': [17, '', '']}
```

Having to do different things for the root and other vertices, and having to name vertices, is quite annoying. The former issue can be fixed with special case code in insert, but the latter is more fundamental to the dictionary representation, as discussed earlier.

Finally, let's see how we can obtain the list of numbers stored in a BST in sorted (ascending) order. In-order traversal exploits the BST property to produce

a sorted list, as shown in the following code.

```
1.  def inOrder(bst):
2.      outputList = []
3.      inOrderHelper(bst, 'root', outputList)
4.      return outputList

5.  def inOrderHelper(bst, vertex, outputList):
6.      if vertex == '':
7.          return
8.      inOrderHelper(bst, bst[vertex][1], outputList)
9.      outputList.append(bst[vertex][0])
10.     inOrderHelper(bst, bst[vertex][2], outputList)
```

The interesting code is in insertHelper. Lines 6–7 are the base case. In-order traversal means traversing the left child first (line 8), then appending the current vertex's number to the in-order list (line 9), then traversing the right child (line 10).

This leads to yet another sorting algorithm: Given a set of numbers in arbitrary order, insert them into an initially empty BST one at a time. Then use inOrder to obtain a sorted list.

OOP-Style Binary Search Trees

We will reintroduce you to classes and object-oriented programming (OOP) and explain the basics. It's "reintroduce" because we have already used built-in Python classes such as lists and dictionaries, and invoked methods on list and dictionary objects, the essence of OOP.

Here are a few examples. In puzzle 1, we wrote intervals.append(arg), which calls the append method on the list intervals and adds the argument arg to the list intervals. In puzzle 3 we used deck.index(arg) to find the index of the argument element arg in the list deck of cards. And in puzzle 14 we used vset.remove(arg) to remove the argument element arg from the set vset.

The big difference in this puzzle is that we will define our own Python classes. We'll begin with defining a vertex class that will correspond to a vertex in a BST for our purposes, but could easily be used in graphs or other kinds of trees.

```
1.  class BSTVertex:
2.      def __init__(self, val, leftChild, rightChild):
3.          self.val = val
```

```
4.              self.leftChild = leftChild
5.              self.rightChild = rightChild

6.          def getVal(self):
7.              return self.val
8.          def getLeftChild(self):
9.              return self.leftChild
10.         def getRightChild(self):
11.             return self.rightChild
12.         def setVal(self, newVal):
13.             self.val = newVal
14.         def setLeftChild(self, newLeft):
15.             self.leftChild = newLeft
16.         def setRightChild(self, newRight):
17.             self.rightChild = newRight
```

Line 1 defines the new class BSTVertex. Lines 2–5 define the class's constructor, which needs to be named __init__. The constructor not only creates a new BSTVertex object and returns it (there is an implicit return) but also initializes the fields in BSTVertex. The fields themselves are defined through the initialization. Our BSTVertex has three fields: a value val, a left child leftChild, and a right child rightChild. We could easily have added a name field but chose not to, partly to contrast with the dictionary representation and partly to show that it is not required.

Here's how we construct a BSTVertex object:

```
root = BSTVertex(22, None, None)
```

Note that we did not call __init__ but rather called the constructor by the name of the class BSTVertex, and we only specified three arguments, corresponding to the last three arguments of __init__. The argument self is just included so we can refer to the object inside the procedure. Otherwise, we would be writing leftChild = leftChild, which would be *very* confusing both to the reader and to Python! In the above constructor call we have created a vertex root (think of it as the root vertex of a BST that is yet to be created) that has a number/value 22 and no children.

In lines 6–17 we define methods to access and modify (or mutate) BSTVertex objects. Strictly speaking, these methods are not necessary, but they are good practice in OOP. It is certainly possible to access the value of a vertex bn simply by writing bn.val, rather than n.getVal(). Or to modify the value of the vertex by writing n.val = 10 as opposed to n.setVal(10). Note that we do not have to specify

the argument self in the accessor and mutator methods, similar to how we did not specify it in the constructor method. Methods that read and return values of objects are called accessor methods and methods that modify or mutate values of objects are called mutator methods.

Let's now look at the BST class. Here's part of it:

```
 1.  class BSTree:
 2.      def __init__(self, root):
 3.          self.root = root
 4.      def lookup(self, cVal):
 5.          return self.__lookupHelper(cVal, self.root)
 6.      def __lookupHelper(self, cVal, cVertex):
 7.          if cVertex == None:
 8.              return False
 9.          elif cVal == cVertex.getVal():
10.              return True
11.          elif (cVal < cVertex.getVal()):
12.              return self.__lookupHelper(cVal,\
12a.                                 cVertex.getLeftChild())
13.          else:
14.              return self.__lookupHelper(cVal,\
14a.                                 cVertex.getRightChild())
```

The constructor for the BST is very simple and is defined in lines 2–3. It creates a BST with a root vertex that is specified as an argument. The assumption is that we will use a BSTVertex object as a root, though that is not specified in the constructor. However, the lookup method specified in lines 4–14 makes this clear. The procedure lookup performs the search in the same way as the dictionary-based lookup we showed you earlier, but works on fields of objects rather than positions in lists or dictionaries. We are following Python convention in adding __ prefixes to the names of functions and procedures, such as lookupHelper, that are not called directly by the user of the class.

Here's how we can create a BST and look up vertices.

```
root = BSTVertex(22, None, None)
tree = BSTree(root)
lookup(tree.lookup(22))
lookup(tree.lookup(14))
```

The first lookup returns **True**, and the second, **False**. This isn't a particularly interesting BST, so let's look at how we can insert vertices into the BST. Note

that the code below is inside the BSTree class, so we continue the line numbering. The indentation of insert is at the same level as lookup.

```
15.     def insert(self, val):
16.         if self.root == None:
17.             self.root = BSTVertex(val, None, None)
18.         else:
19.             self.__insertHelper(val, self.root)

20.     def __insertHelper(self, val, pred):
21.         predLeft = pred.getLeftChild()
22.         predRight = pred.getRightChild()
23.         if (predRight == None and predLeft == None):
24.             if val < pred.getVal():
25.                 pred.setLeftChild((BSTVertex(val, None, None)))
26.             else:
27.                 pred.setRightChild((BSTVertex(val, None, None)))
28.         elif (val < pred.getVal()):
29.             if predLeft == None:
30.                 pred.setLeftChild((BSTVertex(val, None, None)))
31.             else:
32.                 self.__insertHelper(val, pred.getLeftChild())
33.         else:
34.             if predRight == None:
35.                 pred.setRightChild((BSTVertex(val, None, None)))
36.             else:
37.                 self.__insertHelper(val, pred.getRightChild())
```

This code is much cleaner than the dictionary-based code. The procedure insert handles the case where the root of the tree does not exist. In fact, now that we have insert, we can do this:

```
tree = BSTree(None)
tree.insert(22)
```

This will create a BST with a root that has the number 22 and no children, and avoids having to create a BSTVertex object for the root before the BSTree object is created. And if you don't feel like typing **None** when you create empty trees, you can modify the tree constructor to have a default parameter:

```
2.     def __init__(self, root=None):
3.         self.root = root
```

Finally, here's a procedure that does in-order traversal of a tree. Again, it uses

the exact same algorithm as the dictionary-based representation. This code, too, is inside the BSTree class.

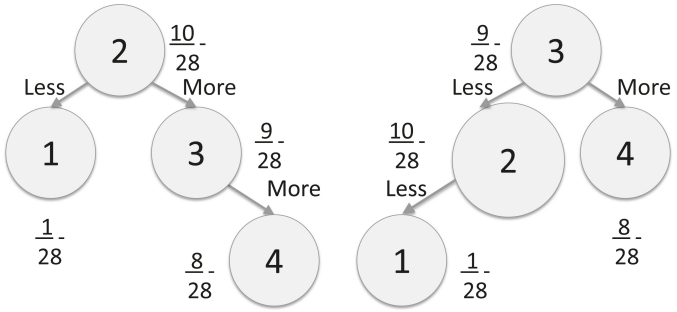
```
38.     def inOrder(self):
39.         outputList = []
40.         return self.__inOrderHelper(self.root, outputList)

41.     def __inOrderHelper(self, vertex, outList):
42.         if vertex == None:
43.             return
44.         self.__inOrderHelper(vertex.getLeftChild(), outList)
45.         outputList.append(vertex.getVal())
46.         self.__inOrderHelper(vertex.getRightChild(), outList)
47.         return outList
```

The best part of this OOP-style code is that it is much more easily augmented. If we want to add a name to our BST vertices, we can easily do that by adding a field to BSTVertex. You will extend the BST data structure in exercises 1 and 2.

Back to Our Puzzle: Algorithm

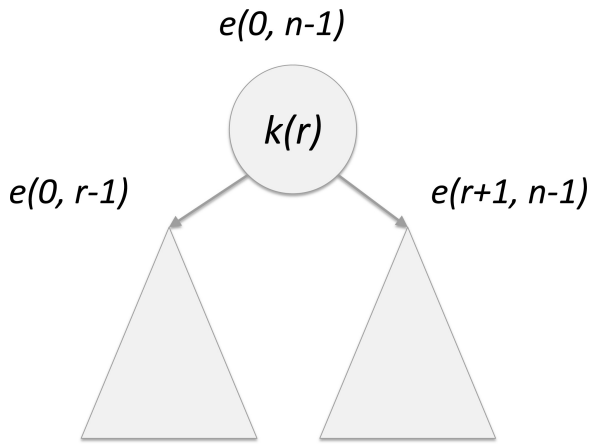
Now that we have the data structure we need, let's try a greedy algorithm to solve our problem. A greedy algorithm chooses as the root of the BST the number with the highest probability. The reasoning is that you want the smallest depth for the number with the highest probability. Once a root is chosen, we know what numbers need to be to the left of the root, and what numbers need to be to the right. We apply the greedy rule again to each of the sides. This seems like a fine algorithm that should work well. It does in many cases. Unfortunately, it does not produce the optimal BST in many others.[1] An example where it fails is shown below.



Let's assume the probability that Thinker chooses 1, 2, 3, or 4 is $\frac{1}{28}$, $\frac{10}{28}$, $\frac{9}{28}$, and $\frac{8}{28}$,

respectively. The probability that Thinker chooses 2 is the highest, so a greedy algorithm would choose it as the root. This means that 1 needs to be to the left, and 3 and 4 need to be to the right of the root. Since choosing 3 has a higher probability than choosing 4, we choose 3 as the first vertex on the right side of the original root choice. This gives us the BST on the left (diagram above), which produces an average number of guesses or weight equaling $\frac{1}{28} \cdot 2 + \frac{10}{28} \cdot 1 + \frac{9}{28} \cdot 2 + \frac{8}{28} \cdot 3 = \frac{54}{28}$. There is a better BST, that is, one with a smaller weight, as shown on the right. This BST chooses 3 as the root and corresponds to a weight of $\frac{1}{28} \cdot 3 + \frac{10}{28} \cdot 2 + \frac{9}{28} \cdot 1 + \frac{8}{28} \cdot 2 = \frac{48}{28}$.

This means we have to try different possibilities for the root vertex and choose the best root to minimize the weight. Suppose $e(0, n - 1)$ is the minimum weight, given numbers $k(0)$, $k(1)$, …, $k(n - 1)$. Denote the probability of each $k(i)$ as $p(i)$. We will assume $k(0) < k(1) \ldots < k(n - 1)$. This means that if we choose $k(i)$ as the root, $k(0)$, … $k(r - 1)$ will be to the left of this root, and $k(r + 1)$, … $k(n - 1)$ will be to the right, as shown in the picture below. Both of the subtrees (shown as triangles) are also BSTs and have minimum weights $e(0, r - 1)$ and $e(r + 1, n - 1)$.



The overall recursive decomposition is given by the two equations below, where i is the start index and j is the end index.

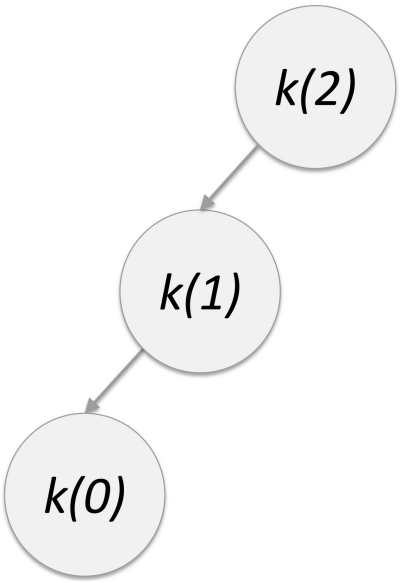
$$e(i, i) = p(i)$$

$$e(i, j) = \min_{r=i}^{j}(e(i, r-1) + e(r+1, j)) + \sum_{s=1}^{j} p(s)$$

If there is a single number i, there is only one BST possible, and its minimum weight is $p(i)$. This is the base case.

If we have a set of numbers $k(i)$ through $k(j)$, we have to choose every possible root and select the minimum weight corresponding to the best root choice. There is an additional summation term in the second equation that warrants explanation. This term accomplishes the multiplication of $p(i)$ by $(D(i) + 1)$ in our weight equation, $\Sigma_i Pr(i) \cdot (D(i)+1)$. For each vertex i, $p(i)$ is added each time the vertex is not selected as a root, and added one last time when the vertex is selected as a root, giving it a weight of $(D(i) + 1)$. It's worth emphasizing that once the vertex i is selected as a root, it does not belong to either subtree in the recursive decomposition, so $p(i)$ is not added again.

To see this clearly, consider three numbers $k(0)$, $k(1)$, and $k(2)$ whose optimal BST is shown below.



The equations are:

$$e(0,2) = e(0,1) + \cancel{e(3,2)} + p(0) + p(1) + p(2)$$
$$e(0,1) = e(0,0) + \cancel{e(2,1)} + p(0) + p(1)$$
$$e(0,0) = p(0)$$

Substituting values produces:

$$e(0, 2) = 3 \cdot p(0) + 2 \cdot p(1) + p(2)$$

Exactly what we want.

Code to Solve Our Puzzle

First, we show the main procedure that provides the high-level structure for the code to solve the puzzle.

```
1.    def optimalBST(keys, prob):
2.        n = len(keys)
3.        opt = [[0 for i in range(n)] for j in range(n)]
4.        computeOptRecur(opt, 0, n-1, prob)
5.        tree = createBSTRecur(None, opt, 0, n-1, keys)
6.        print('Minimum average # guesses is', opt[0][n-1][0])
7.        printBST(tree.root)
```

Line 3 initializes the data structure opt that stores the minimum weights for the different subproblems associated with subtrees. We need a two-dimensional list since we have to store $e(i, j)$ values. Each element of the opt list is a 2-tuple: The first element is the $e(i, j)$ value and the second is the choice of root that produced this value. The index i of the number keys[i], corresponding to the chosen root, is stored as the second element.

The procedure computeOptRecur computes these optimal solutions and fills in the opt list (line 4). Then, we have to turn these values into the optimal BST (just like we had to trace back to find the coins in puzzle 18, our coin selection puzzle). This is done by procedure optimalBSTRecur (line 5).

Here's how the computation of optimal weights is done recursively.

```
 1.    def computeOptRecur(opt, left, right, prob):
 2.        if left == right:
 3.            opt[left][left] = (prob[left], left)
 4.            return
 5.        for r in range(left, right + 1):
 6.            if left <= r - 1:
 7.                computeOptRecur(opt, left, r - 1, prob)
 8.                leftval = opt[left][r-1]
 9.            else:
10.                leftval = (0, -1)
11.            if r + 1 <= right:
12.                computeOptRecur(opt, r + 1, right, prob)
13.                rightval = opt[r + 1][right]
14.            else:
15.                rightval = (0, -1)
16.            if r == left:
17.                bestval = leftval[0] + rightval[0]
18.                bestr = r
```

```
19.                 elif bestval > leftval[0] + rightval[0]:
20.                     bestr = r
21.                     bestval = leftval[0] + rightval[0]
22.             weight = sum(prob[left:right+1])
23.             opt[left][right] = (bestval + weight, bestr)
```

Lines 2–4 correspond to the base case of a single number. Lines 5–23 correspond to the recursive case, where we have to choose different numbers as roots and select the root that produces the minimum weight.

Lines 6–10 make a recursive call, provided the left subtree has at least two numbers for the choice of root. Similarly, lines 11–15 make a recursive call, provided the right subtree has at least two numbers.

Lines 16–23 find the minimum weight value. Lines 16–18 initialize bestval in the first iteration of the loop, and lines 19–21 update bestval if we have found a solution with smaller weight. Finally, lines 22–23 add in the summation term $\Sigma_{s=i}^{j} p(s)$ and update the opt list.

You might have observed that the code for computeOptRecur is amenable to memoization, and you are right. You get to memoize this code in exercise 3.

Let's now look at the procedure that creates the optimal BST once we know the optimal weights for all the subtrees.

```
1.  def createBSTRecur(bst, opt, left, right, keys):
2.          if left == right:
3.              bst.insert(keys[left])
4.              return bst
5.          rindex = opt[left][right][1]
6.          rnum = keys[rindex]
7.          if bst == None:
8.              bst = BSTree(None)
9.          bst.insert(rnum)
10.         if left <= rindex - 1:
11.             bst = createBSTRecur(bst, opt, left, rindex - 1, keys)
12.         if rindex + 1 <= right:
13.             bst = createBSTRecur(bst, opt, rindex + 1, right, keys)
14.         return bst
```

This procedure assumes that the given list of numbers in keys has length at least 2. The procedure looks at the choice of root for the current problem (line 5). It creates a BST with the chosen root (lines 7–9) and handles the case of when the BST does not exist. Recursive calls are made for the left (lines 10–11) and right

(lines 12–13) subtrees, provided they are not empty.

Lines 2–4 are the base case where the BST has a single vertex. A base case is only encountered after the BST has been created, since we assume there are at least two numbers in the original BST. Therefore, we do not need to check whether the BST exists.

Let's take a look at how we can print the BST in text form.

```
1.  def printBST(vertex):
2.      left = vertex.leftChild
3.      right = vertex.rightChild
4.      if left != None and right != None:
5.          print('Value =', vertex.val, 'Left =',
                    left.val, 'Right =', right.val)
6.          printBST(left)
7.          printBST(right)
8.      elif left != None and right == None:
9.          print('Value =', vertex.val, 'Left =',
                    left.val, 'Right = None')
10.         printBST(left)
11.     elif left == None and right != None:
12.         print('Value =', vertex.val, 'Left = None',
                    'Right =', right.val)
13.         printBST(right)
14.     else:
15.         print('Value =', vertex.val,
                    'Left = None Right = None')
```

Note that the procedure takes the root of the BST (i.e., a BSTVertex) as its argument. This way, it can simply call itself recursively on the left and right children of the root.

Let's run the code on our original motivating example to see if it produces the right BSTs.

```
keys = [1, 2, 3, 4, 5, 6, 7]
pr = [0.2, 0.1, 0.2, 0.0, 0.2, 0.1, 0.2]
optimalBST(keys, pr)
```

This produces:

```
Minimum average # guesses is 2.3
Value = 3 Left = 1 Right = 5
Value = 1 Left = None Right = 2
Value = 2 Left = None Right = None
Value = 5 Left = 4 Right = 7
```

```
Value = 4 Left = None Right = None
Value = 7 Left = 6 Right = None
Value = 6 Left = None Right = None
```

Let's next run the code on the example where the greedy algorithm fails.

```
keys2 = [1, 2, 3, 4]
pr2 = [1.0/28.0, 10.0/28.0, 9.0/28.0, 8.0/28.0]
optimalBST(keys2, pr2)
```

This produces:

```
Minimum average # guesses is 1.7142857142857142
Value = 3 Left = 2 Right = 4
Value = 2 Left = 1 Right = None
Value = 1 Left = None Right = None
Value = 4 Left = None Right = None
```

Comparing Data Structures

We have seen lists, dictionaries, and BSTs among the many data structures in this book. Lists are the simplest of the three, and require the least storage. If you have a collection of items, and all you want to do is store them and process them sequentially, the list is perfect for the job. However, lists can be inefficient for many tasks, such as checking for membership, where the required number of operations grows with the length of the list.

Dictionaries generalize lists in their ability to be indexed, but also provide an efficient means of querying membership. Using a hash table as the underlying data structure means that looking up an item requires only a few operations. On the other hand, range queries such as "Do I have a key between x and y?" require enumerating all the keys in the dictionary because dictionaries do not keep items in sorted order.

Looking up a key in a BST requires $\log n$ operations and is not as efficient as a dictionary, but is significantly more efficient than a list. Range queries on keys are conveniently implemented on BSTs since BSTs are a sorted representation—see exercise 4.

Choosing the right data structure for the task at hand often leads to interesting algorithmic puzzles!

Exercises

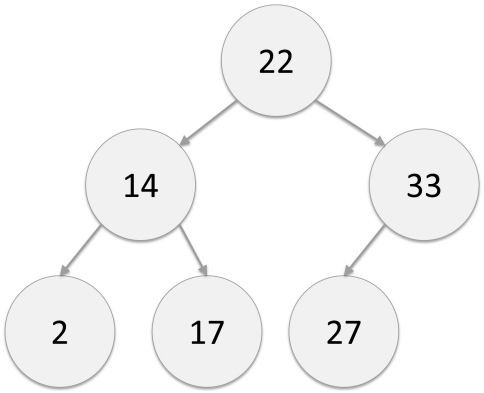
Exercise 1: Add a `getVertex` method to the `BSTree` class to return a `BSTVertex`,

rather than **True** or **False** like lookup does. This is a modification of the BST data structure and has nothing to do with the puzzle per se.

Exercise 2: Add a size method mimicking inOrder that computes the size of the BST, which is defined as the number of vertices in the BST—another modification of the BST data structure.

Exercise 3: Memoize computeOptRecur, which does a huge amount of redundant work to create computeOptRecurMemoize. The memoized procedure should only compute each opt entry (i.e., $e(i, j)$) *once*.

Exercise 4: Implement a procedure rangeKeys(bst, k1, k2) that determines all keys k in the BST such that k1 <= k <= k2, and prints them out in increasing order. For the BST b below, rangeKeys(b, 10, 22) should print 14, 17, and 22.



Note

1. This keeps happening! Notwithstanding puzzle 16, greed is mostly bad.

Index