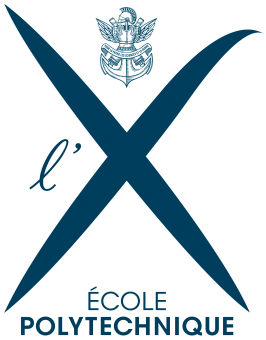
CSE 102: COMPUTER PROGRAMMING

LECTURE NOTES

Kaustuv Chaudhuri
kaustuv.chaudhuri@inria.fr

VERSION OF SPRING 2019



2019-06-04

CONTENTS

PREFACE

This document represents the syllabus for the second semester introduction to **computer programming** course in the bachelor program at the École polytechnique, France. This course:

- targets all first year students regardless of major;
- focuses on **computer science** topics; and
- uses **Python 3.0** as the implementation language.

This course assumes that the reader is already familiar with basic Python 3.0.

Acknowledgements Thanks to the following students for finding bugs and helping to improve the quality of this text: Amine Abdeljaoued, María Cano, Archit Chaturvedi, Maxence Dulac, Matthieu Melennec, Ashish Nayak, Tarcisio Soares Teixeira Neto.

Thanks also to François Bidet, Uli Fahrenberg, Anna Pazii, and Pierre-Yves Strub for help with editing the text and with writing and administering the accompanying tutorials.

1 ADVANCED PYTHON PROGRAMMING

This chapter is designed to extend and round out your knowledge of the Python programming language.

1.1 EXCEPTIONS AND CONTEXT MANAGERS

Exceptions and raising A computation in Python normally proceeds top to bottom according to control structures such as conditionals and loops. This normal process of computation makes certain assumptions about the inputs and attempts to guarantee certain outcomes. However, in the course of the computation if either the assumptions are detected to be unmet or if some outcome is determined to be impossible, then the computation would need to be aborted. One way to abort a program to simply cause the computer to turn itself off, or, less drastically, to cause Python to exit. However, many errors are not so catastrophic that they require such extreme measures, and good code is often written *defensively* to make sure that many predictable abnormalities have countermeasures. This is where Python uses *exceptions*.

Exceptions are a kind of object that can be *raised* in computation that needs to abort. When an exception is raised, Python enters a special execution mode that attempts to break out of computations one by one. The currently executing code (usually a function) is aborted and the exception is propagated higher in chain of function calls that led up to the exception. When the exception propagates to the next level higher in the call chain, it aborts *that* call and then propagates one level higher; and then aborts it and propagates one level higher, and so on. This chain of aborted execution followed by upward propagation of the exception continues until it reaches a point in the code where that exception can be *caught*, at which point the special execution mode ends and ordinary execution resumes.

When you are running Python code in an interpreter like `python3` or `ipython`, any exceptions that get propagated all the way to the top are then caught by Python itself and turned into an error message. We can use this to see a number of exceptions that are already built into Python. (You have no doubt encountered such error messages yourself.) Here are some examples.

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

You can also raise any exception you want yourself with the following kind of statement.

```
raise obj
```

Here, the object `obj` must be an instance of the class `Exception`, or one of its derived classes. This exception object is the *payload* that will be sent along as the exception propagates and delivered to any code that catches the exception. Most standard exceptions in Python have names that end with `Error`, as you can see

in the interactive trace above.[1] You can use these exceptions if you want, but because they already have specific meanings in Python it is not a good idea to reuse them. For instance, if you use `ZeroDivisionError` in your code for situations that have nothing to do with division by zero, you may confuse parts of the Python standard library for dealing with zero denominators. That being said, there are certain standard exceptions that are actually intended to be **`raise`**d outside the standard library by user code; usually these exception classes *don't* end in `Error`. The most common example is the `StopIteration` exception that we will see below in Section 1.2.

Defining an `Exception` subclass for use in your own code is straightforward. To give a slightly farcical example, imagine that your code can encounter *"pocket monsters"* and you want to signal them up the call chain with a `PocketMonsterEncounter` with the monster object as a payload. Defining it is trivial:

```
class PocketMonsterEncounter(Exception):
    pass
```

(You can replace the trivial body **`pass`** with a docstring if you want to have good code hygiene.) Then, in code that encounters pocket monsters, you can just **`raise`** this exception instead. For example:

```
def movement_encounter(thing):
    if instanceof(thing, Grass):
        # handle grass, which lets the player through
    elif instanceof(thing, Rock):
        # handle rock, which blocks the player
    elif instanceof(thing, PocketMonster):
        # pocket monster encounters are different!
        ex = PocketMonsterEncounter(thing) # create the exception object
        raise ex
        # the previous two lines are generally combined into:
        #     raise PocketMonsterEncounter(thing)
    else:
        # other code
```

CATCHING AND PROCESSING EXCEPTIONS To catch and process exceptions (also known as *handling* in other languages and in the general programming vernacular), you have to mark the code that could potentially **`raise`** an exception with a new kind of control structure called the **`try`** form.

Figure 1.1 shows the fully general form of the **`try`** construct. Following the **`try`** block header that protects a block of code that could raise an exception, there is a sequence of **`except`** clauses that indicate the exceptions they are trying to catch and the code to run when they do catch an exception. For instance, the code that calls `movement_encounter()` can catch the `PocketMonsterException` as follows:

```
try:
    movement_encounter(thing)
except PocketMonsterEncounter: # catch them all!
    print('Encountered a pocket monster. Time to battle!')
    start_battle()
```

This form simply detects that the caught exception is an instance of the `PocketMonsterEncounter` class. We may also want to get the exception object itself to extract its payload. For that, we have to assign a variable to bind to that object as follows.

[1] For a complete list see: `https://docs.python.org/3/library/exceptions.html`

```
try:
    # code that can raise exceptions
except Exc1:
    # handler for exception Exc1
except Exc2 as var:
    # handler for exception Exc2, with the exception object
    # (the argument of raise) held in variable var
...
except:
    # optional default handler for any exceptions not otherwise caught
else:
    # optional block to run if no exceptions were raised
finally:
    # optional block with cleanup actions
```

Figure 1.1: The general form of the `try` construct

```
try:
    movement_encounter(thing)
except PocketMonsterEncounter as pm_ex: # catch them all!
    # pm_ex.args is a *tuple* of all the arguments to the
    # initializer of the exception object
    pm = pm_ex.args[0]
    print('Encountered {}. Time to battle!'.format(pm))
    start_battle_with(pm)
```

There can be multiple handlers for the same `try` block. For instance, if we think that there is a chance of a division by zero in the `movement_encounter()` function, then we could handle that as well:

```
try:
    movement_encounter(thing)
except PocketMonsterEncounter as pm_ex:
    # same as before
except ZeroDivisionError as ex:
    print('BUG in movement code: {}'.format(ex))
    raise ex # re-raise the exception to higher levels
```

The `except` clauses are tried top to bottom, so if there are multiple handlers that can catch an exception the topmost one wins. This is usually the case when we have handlers for both an exception class and its superclasses such as `Exception` itself. We can thus catch entire classes of exceptions, and even catch *all* exceptions if we want:

```
try:
    movement_encounter(thing)
except PocketMonsterEncounter as pm_ex:
    # same as before
except ZeroDivisionError as ex:
    # same as before
except Exception as ex:
    print('Exception with no other handler: {}'.format(ex))
    raise ex
```

Catching all exceptions this way is generally a bad idea (unless you immediately re`raise` the exception) since it can hide important bugs in other parts of your code. You should narrowly tailor your handlers to exactly those exceptions that you are *prepared to recover from*.

Like with `for` and `while` loops, there is an optional `else` block that comes after all the `except` blocks in case no exceptions were raised in the code protected with `try`. Lastly, there is an optional `finally` block that contains code that is always executed at the end no matter which code path is taken (or not taken) through the `except` and `else` blocks. This is primarily used to perform *cleanup* steps that must be taken no matter what. For instance, if we're making a telephone call, we could write code such as the following:

```
1  try:
2      call = phone_number(num, person)
3      converse_with(person, call)
4      return True
5  except VoiceMail:
6      print('{} was not available'.format(person))
7      leave_message(call)
8      return False
9  except WrongNumber:
10     print("{} is not {}'s number".format(num, person))
11     raise IncompleteCall
12 finally:
13     hang_up(call) # always hang up the call
```

Be mindful that as soon as a `try` block is entered, its `finally` block is guaranteed to eventually run. Even if the `try` block or any of the other `except` and `else` blocks jump out in any other way such as with `break`, `continue`, `return` (such as in lines 4 and 8), or even other `raise`s (such as in line 11), the `finally` block still runs before the jump happens. In other words, line 13 above will always run.

Context Managers The `try ... finally` protocol is a very versatile mechanism for running computations with guaranteed set-up and tear-down steps. However, since `try` blocks are generally associated with exceptions, the intent of the code that uses `finally` for a tear-down procedure may be obscure. Python therefore provides an alternative and more object oriented mechanism for such *delimited* computations in the form of *context managers*.

A context manager is any object that implements the special methods `__enter__()` and `__exit__()`. The `__enter__()` optionally returns an object that can be used to represent a resource that is being protected, which will eventually be properly disposed off by the `__exit__()` method. This resource can really be anything: e.g., an open file, a network communication stream, a window being displayed, a handle to a critical section of your program, and so on.

To use a context manager, you would use the `with ... as` control structure that has this form:

```
with expr as v:
    # code that uses v
```

Here, `expr` is some expression that evaluates to a context manager object, whose `__enter__()` method is run and its result bound to the variable `v`. The body of the `with` block then runs the code that operates on `v`. When the block exits for whatever reason (normal completion, jumps, exceptions, etc.), the `__exit__()` method of the context manager is run just before the exit. If the variable `v` is not important, then the `as v` clause can be omitted.

1.1.1 *Exercises*

1. You are given the following function that makes many kinds of assumptions about its input.

```
1  def carefree_f(seq):
2      (u, v) = seq.pop()
3      while seq:
4          (x, y) = seq.pop()
5          u += x
6          v += y
7      return u / v
```

   Wrap a call to `carefree_f(seq)` in a `try ... except` structure to catch as many possible exceptional cases for seq as you can think of. Some possibilities (not exhaustive!) to consider: (1) `seq` is empty, (2) the value of `v` in line 7 is 0, and (3) some element of `seq` is not a 2-tuple of numbers.

2. Define a class for *logs* that will send all its messages both to the standard output (using `print()`) and to a certain file that is part of the initializer of the class. Here is its skeleton.

```
1  class Log:
2      def __init__(self, logfile):
3          self._logfile = logfile
4          self._loghandle = None # file is not open by default
5
6      def log(self, msg):
7          if self._loghandle is None:
8              raise RuntimeError('Invalid use of the Log class')
9          print(msg)
10         self._loghandle.write(msg)
```

   Add the `.__enter__()` and `.__exit__()` member functions to this class so that the following code runs and behaves as expected.

```
1  with Log('log.txt') as lf:
2      lf.log('Hello, logging world!')
3      raise RuntimeError
```

   The `.__enter__()` function should open the log file in mode `'a'` in order to append to it. Make sure that the file handle is closed in the `.__exit__()` function so that even though line 3 above raises an exception the file handle is still disposed of properly.

1.2 CONTAINERS

A *container* is an object whose primary purpose is to contain other objects. Python has many built in sorts of containers: lists, sets, dictionaries, and even implicit containers[2] such as those returned by the `range()` function. Python has built in syntax and several functions to simplify building and operating with containers.

[2] Such containers behave as if they contain the objects but don't necessarily have the objects allocated and stored in memory.

As our running example, let us consider a 3-dimensional scatter plot consisting of values assigned to (x, y) coordinates, which is internally represented as a dictionary `_data`. (Recall the convention that this field is intended to be *private* to the class because its name starts with an underscore.)

```
1  class Scatter3D:
2      def __init__(self):
3          self._data = {}
4
5      def insert(self, x, y, v):
6          self._data[(x, y)] = v
```

QUERYING Some of the most basic operations that one would perform on such a collection is to get its size and get particular elements. Canonically, the size of any container is given by the built in `len()` function which translates a call `len(s)` to the method call `s.__len__()`. So, let us implement this method.

```
7      def __len__(self):
8          return len(self._data)
```

Note that we are merely delegating to the `__len__()` method of the underlying dictionary.

Next, we may want to check if the point `(x, y)` has been assigned a value in the plot. Membership queries in Python are done with the keywords `in` and `not in`, both of which are internally transformed by Python. Precisely, `(x, y) in s` is transformed to `s.__contains__((x, y))` and `(x, y) not in s` to `not s.__contains__((x, y))`.

```
9      def __contains__(self, key):
10         return key in self._data
```

We may also want to query the value of a given point `(x, y)`. We could do this by using `_data` directly, but that would violate the API: external code is not supposed to have direct access to the private data attributes of an object. We instead want to enrich the class so that the values of a `Scatter3D` object `s` can be queried as `s[x, y]`. Internally, Python would transform `s[x, y]` to the member function call `s.__getitem__((x, y))`, so we can implement this function in our class.

```
11     def __getitem__(self, pt):
12         return self._data[pt]
```

MODIFYING There is already a function for inserting data to the plot using `.insert()`, but we may want to use the assignment notation `s[x, y] = v`. Internally this is transformed to `s.__setitem__((x, y), v)`, which means that we have to implement the following function.

```
13     def __setitem__(self, pt, value):
14         self.insert(pt[0], pt[1], value)
```

Deleting data follows the same pattern: `del(s[x, y])` is transformed to `s.__delitem__((x, y))`, so:

```
13     def __delitem__(self, pt):
14         del(self._data[pt])
```

ITERATION A container is said to be *iterable* if it supports iteration with a `for` loop, which in turn allows the object to turned into a list, set, etc. with functions like `list()`, `set()`, etc. Any iterable object must have the member function `__iter__()` which returns a brand new object called the *iterator*. This iterator object must itself have two member functions: `__iter__()` which should return the same object,[3] and `__next__()` which gives the next item in the iteration.

Let us implement iterators in terms of a new class, `Scatter3DIterator`. It is initialized with the items of the private `_data` attribute.

```
15      def __iter__(self):
16          return Scatter3DIterator(self._data.items())
```

The iterator class itself is as follows.

```
1   class Scatter3DIterator:
2       def __init__(self, items):
3           self.data = list(items)
4
5       def __iter__(self):
6           return self
7
8       def __next__(self):
9           if len(self.data) is 0:
10              raise StopIteration
11          ((x, y), v) = self.data.pop()
12          return (x, y, v)
```

The `StopIteration` exception is raised (line 10) when there are no elements remaining to iterate over.

We can now iterate over all the points in the plot `s` as in the following example.

```
>>> for (x, y, v) in s:
...     print('({}, {}): {}'.format(x, y, v))
...
```

1.3 GENERATORS

As we have seen in the Section 1.2, Python supports iteration over the contents of any object that implements the `__iter__()` and `__next__()` member functions. Sometimes, however, we would like to iterate over a sequence of data values that is very long (possibly infinite), or where elements of the sequence are produced as a result of a computation. In such cases, computing and loading the entire collection into our computer's memory may be unduly expensive or even impossible. Of course we can get around this by defining our own classes with bespoke iterators, but a more conceptually lightweight mechanism is available in Python 3: *generators*.

BUILDING A GENERATOR: `return` VS. `yield` A generator object is an iterable object that *encodes* a computation. What does that mean? The standard way to define a computation in Python is to write a *function* that runs a block of Python code that eventually produces an answer that is `return`ed to the caller of the function. For instance, here is a function that immediately returns its argument:

[3] This is needed for the specific way in which `for` loops are implemented.

```
def give_back(k):
    return k
```

```
>>> give_back(42)
42
```

If we change the `return` keyword to `yield`, we see something different.

```
def yield_back(k):
    yield k
```

```
>>> yield_back(42)
<generator object yield_back at 0x7f8169c20840>
```

We get a *generator object*. To get an idea of what this object does, we can use the built in `help()` function to read its docstring.[4]

```
>>> help(yield_back(42))
Help on generator object:

yield_back = class generator(object)
 |  Methods defined here:
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __next__(self, /)
 |      Implement next(self).
```

This object implements `__iter__()` and `__next__()`: hence, it is an iterator. What values does it produce? Well, any finite iterable can be turned into a list using the built in `list()` function,[5] so we can use it to examine the contents of this generator object.

```
>>> list(yield_back(42))
[42]
```

Unsurprisingly, it contains just the singleton `42`.

What sets `yield` apart from `return`, however, is that the generator object can produce multiple answers! For instance, we can try to `yield` the argument three times.

```
def yield_back(k)
    yield k
    yield k
    yield k
```

If we were to have used `return` instead of `yield`, then the second and third `return` would be dead code: no execution path would ever reach them. However, `yield` does not stop the rest of the function from executing; indeed, all three `yield`ed values are produced by iterating over the generator object.

```
>>> list(yield_back(42))
[42, 42, 42]
```

[4] There are other member functions that are not relevant here, so we elide them from the output of `help()`.

[5] This is actually the initializer of the built in `list` class of Python.

Thus, the body of a *generator function* such as `yield_back()` is arbitrary Python code that calls **`yield`** several times, and each of these **`yield`**ed values are produced by the iterator built from calling the generator function. This arbitrary code can have local variables, loops, iterations, IO, etc.[6] For example, here is a variant of **`range`**`()` that **`yield`**s numbers from 0 to its argument.

```
1  def yield_range(k):
2      i = 0
3      while i < k:
4          yield i
5          i += 1
```

```
>>> list(yield_range(5))
[0, 1, 2, 3, 4]
>>> list(yield_range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

EXECUTION MODEL OF A GENERATOR How does the `yield_range()` function above work? Suppose we run `r = yield_range(10)`. This call to `yield_range` executes all the way to line 4 and then stops *just before* the point of executing the **`yield`** statement, which becomes the *generator object* represented by `r`. You can think of `r` as a *suspended block of Python code* with a *current statement pointer* depicted below with a red arrow, with the values of the relevant variables to the left of the arrow.

```
      i = 0
      while i < k:
i:0
k:10  >     yield i
            i += 1
```

This generator object `r` is an iterator (recall: all generators are iterators), and therefore the next element in the iterator can be obtained with the built in **`next`**`()` function.

```
>>> next(r)
0
```

Here's how that works. First the expression in the **`yield`** statement is executed to compute a value which will be produced by the **`next`**`(r)`. In addition to producing the **`yield`**ed value, a side-effect of calling **`next`**`(r)` is to run its code past the indicated **`yield`**, to the end of the loop, and then back to the next **`yield`** statement, at which point the execution gets suspended again just before the **`yield`** keyword.

```
      i = 0
      while i < k:
          yield i
          i += 1
```

As indicated, after the **`yield`** statement the value of `i` goes up by 1 as a result of running line 5. This brings us to the end of the **`while`** loop, so the next iteration begins and we end back at the **`yield`** statement. The execution in the generator object then pauses again right before executing the **`yield`** statement. We can represent this updated state as follows:

[6] In fact, it can also contain **`return`** statements – we'll come back to that later.

```
        i = 0
        while i < k:
i:1 >       yield i
k:10            i += 1
```

If we call `next(r)` now we would get the result `1`, and then the loop would run once more with `i` now set to `2`. This would continue until the final iteration of the loop with `i` being `9`:

```
        i = 0
        while i < k:
i:9 >       yield i
k:10            i += 1
```

The next time we call `next(r)`, it would return `9` and the loop would end. As there is nothing further to `yield`, the generator would then mark itself as done, so that subsequent calls to `next(r)` would raise the `StopIteration` exception that is used to signal that the iteration is finished.

GENERATORS AS DATA STREAMS Because the values that are `yield`ed by a generator are completely controlled by Python code, generator functions can be used to create a number of sophisticated iterators. The simplest such example is a generator that represents an infinite iteration, say an infinite sequence of `1`s.

```
1  def ones():
2      while True:
3          yield 1
```

A slightly more interesting infinite generator is one that returns the sequence of natural numbers.

```
1  def nats():
2      cur = 0
3      while True:
4          yield cur
5          cur += 1
```

Because generator objects can be used just like other Python objects, one can easily write *generator adaptors*, which are generator functions that take generators as arguments. Here, for instance, is a generator function that takes two generators `g1` and `g2` and generates their point-wise sum.

```
1  def pointwise_sum(g1, g2):
2      while True:
3          x1 = next(g1)
4          x2 = next(g2)
5          yield x1 + x2
```

Here it is in action:

```
>>> os = ones()
>>> ns = nats()
>>> xs = pointwise_sum(os, ns)
>>> [next(xs) for _ in range(10)] # get the first 10 elements of xs
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Generator comprehensions Comprehensions for lists, sets, and dictionaries are a simple way to define a collection using ordinary Python expressions. In a similar way we have *generator comprehensions* (sometimes also called *generator expressions*) that `yield` the elements of the comprehension one at a time.

The syntax of the comprehension is the same as that of list comprehensions, except that instead of list delimiters `[]`, we use ordinary parentheses `()`. For example, here is a generator comprehension that adds `1` to every element of the generator `nats()`.

```
>>> xs = (n + 1 for n in nats())
>>> [next(xs) for _ in range(10)] # get the first 10 elements of xs
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note that if we had tried to use a list comprehension instead of a generator comprehension, we would (eventually) run out of memory.

```
>>> xs = [n + 1 for n in nats()]
# eventually exhausts memory
```

Delegation When writing generator functions or generator comprehensions that operate on other generators, it is sometimes useful to be able to declare that the current generator being defined is going to behave just like a given generator `g`. That is to say, we want to say something like:

```
for x in g: yield x
```

Unfortunately, because `for` loops are not expressions in Python, it is not possible to use the above in a context where a generator expression is expected. This is why Python has added *generator delegation* expressions written using `yield from`: the above `for` loop, written as an expression, is `(yield from g)` (parentheses optional).

As an example of its use, imagine we wanted to write a function `gen_append()` that takes two iterators and generates their elements in sequence. Delegation makes it simple.

```
1 def gen_append(first, second):
2     yield from first
3     yield from second
```

```
>>> xs = gen_append([10, 20, 30], nats())
>>> [next(xs) for _ in range(5)]
[10, 20, 30, 0, 1]
```

1.3.1 *Exercises*

1. Write a generator function `filter_multiples(k, g)` that returns a generator for the same elements as `g` except that all multiples of `k` are filtered out. For instance:

```
>>> g = filter_multiples(2, nats())
>>> [next(g) for _ in range(10)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> g = filter_multiples(5, nats())
>>> [next(g) for _ in range(10)]
[1, 2, 3, 4, 6, 7, 8, 9, 11, 12]
```

2. Write a generator function `primes()` that returns a generator for the sequence of prime natural numbers starting with 2. Use the `filter_multiples()` function from the previous exercise.

```
>>> g = primes()
>>> [next(g) for _ in range(10)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

3. Write a generator function `prefix_sums()` to return a generator for the partial sums of prefixes of the following infinite series.

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

That is, the kth number to be generated should be the value of

$$\sum_{i=0}^{k-1} \frac{(-1)^k}{2k+1}.$$

Analytically, we know that this series converges to $\pi/4$; we can observe this convergence:

```
>>> g = prefix_sums()
>>> [round(abs(math.pi - 4 * next(g)), 4) for _ in range(20)]
[0.8584, 0.4749, 0.3251, 0.2464, 0.1981, 0.1655, 0.1421, 0.1245,
 0.1108, 0.0998, 0.0907, 0.0832, 0.0768, 0.0713, 0.0666, 0.0624,
 0.0588, 0.0555, 0.0526, 0.05]
```

4. *(Acceleration Transform)* Converging sequences like that generated by `prefix_sums()` can be *sped up* by considering a number of consecutive terms and then normalizing their squared error, which makes the modified sequence converge to the same limit faster. One such transformation, due to Aitken, is to take a sequence of numbers $s_0, s_1, \ldots$ and produce a modified sequence $\tilde{s}_0, \tilde{s}_1, \ldots$ where the kth term of the modified sequence is given by:

$$\tilde{s}_k = s_{k+2} - \frac{(s_{k+2} - s_{k+1})^2}{s_{k+2} - 2s_{k+1} + s_k}$$

Write a generator function `accelerate(g)` that returns a generator for the accelerated sequence of numbers generated by `g` using Aitken's transformation above.

```
>>> g = accelerate(prefix_sums())
>>> [round(abs(math.pi - 4 * next(g)), 5) for _ in range(10)]
[0.02507, 0.00826, 0.00365, 0.00191, 0.00112, 0.00071, 0.00048,
 0.00034, 0.00025, 0.00019]
>>> g = accelerate(accelerate(prefix_sums()))
>>> [abs(math.pi - 4 * next(g)) for _ in range(5)]
[0.0005126095681018406, 0.00014243713957640125,
 5.06704064724417e-05, 2.1363388365447378e-05,
 1.0188013049194922e-05]
```

5. The *Pell numbers* are a sequence of numbers $p_0, p_1, p_2, \ldots$ defined by the following equations:

$$p_0 = 0, \qquad p_1 = 1, \qquad p_{k+2} = 2p_{k+1} + p_k \text{ for } k \in \mathbb{N}.$$

Write the generator function `pell()` that returns a generator for the Pell numbers. Use it to build a sequence that converges to $\sqrt{2}$ by returning the sequence of numbers $s_0, s_1, s_2, \ldots$ where:

$$s_k = \frac{p_k + p_{k+1}}{p_{k+1}} \text{ for all } k \in \mathbb{N}.$$

6. *(Puzzle)* With the following definition, can you predict what sequence `f()` generates? (After you think you've figured it out, try it out in Python to see if you got it right.)

```
def f():
    yield 0
    yield 1
    (a, b) = (f(), f())
    next(b)
    while True:
        yield next(a) + next(b)
```

7. Rewrite the Pell number generator in the style of the `f` function of the previous exercise.

1.4 ANONYMOUS FUNCTIONS

When you write a function in Python with a definite purpose, you choose a name and use the **`def`** keyword to define it. You can do this anywhere you can add a definition: inside a module, inside other functions, in **`class`** definitions, and so on. However, sometimes you need to use a one-off function that is so trivial and for a highly specific purpose that coming up with a name for it is almost as hard as writing it. For these cases, Python offers the ability to define *anonymous functions*, sometimes also called *lambda functions*.

The term *lambda* comes from the keyword **`lambda`** that is used to define such anonymous functions. Why *lambda*? Because the mathematical foundation of programming languages that are based around creating, communicating, and computing with functions is the λ-*calculus* (sometimes written out as *lambda calculus*). All programming languages that support a programming style called *functional programming* are fundamentally variations on the λ-calculus. Programming languages that are primarily built for functional programming include LISP, Scheme, Haskell, and OCaml, but many other languages support functional programming to varying degrees. Python too supports functional programming to an extent.

DEFINING ANONYMOUS FUNCTIONS The general form of an anonymous function is:

```
lambda v1, v2, ..., vk: expr
```

Here `v1`, `v2`, etc. are the arguments of the anonymous function, and the body of the function is an expression whose value is implicitly **`return`**ed by the anonymous function. Because the body of an anonymous function is limited to expressions, it cannot have any control structures such as **`if ... else`** blocks, loops, or even **`return`** statements. Moreover, since the function has no name, it is also not possible to make recursive calls with these functions. Anonymous functions are therefore much more syntactically restricted than named functions, to go with their intended one-off usage.

Here is a simple example of an anonymous function that doubles its input argument.

```
>>> lambda x: x * 2
<function <lambda> at 0x7f816968ba60>
```

To use the anonymous function, we could place it immediately in the function position of a function call, i.e., in the position of `f` in the expression `f(42)`.

```
>>> (lambda x: x * 2)(42)
84
```

We could also *assign* the anonymous function to other variables or data attributes, or pass it as an argument to other functions. An anonymous function is itself an *expression*, and hence can be used in any setting where an expression is expected. As an example, here we assign the above function to the variable `double` and then use the variable to make an ordinary function call.

```
>>> double = lambda x: x * 2
>>> double(42)
84
```

The most important use of anonymous functions is as arguments to other functions, which we will illustrate with an example. Consider the built in `min()` function. It can be used in the following form: `min(iterable, key=func)` where the `key` parameter is a function that is applied to every element of the `iterable` in order to obtain a result that is actually used for comparison. For instance, suppose we had a list of pairs of numbers, and we wanted to find the element with the smallest first component; e.g., for the list `[(0, 10), (20, 42), (-10, 157)]` we want to get the last element, `(-10, 157)`. We would use `min()` with a `key` argument that is an anonymous function that simply picks the first component.

```
>>> l = [(0, 10), (20, 42), (-10, 157)]
>>> min(l, key=lambda p: p[0])
(-10, 157)
```

The way it works is that it applies the anonymous function assigned to `key` to each element of the list before comparing it, so instead of comparing `(0, 10)` to `(20, 42)`, it actually compares `key((0, 10))` to `key((20, 42))`, i.e., `0` to `20`. In this way it selects the element that minimizes `key()`, which happens to be the final element `(-10, 157)`.

This is a surprisingly versatile feature of the `min()` function. Suppose instead we wanted the element with the smallest sum of the two components, we could simply modify the `key` parameter to compute the sum instead.

```
>>> min(l, key=lambda p: p[0] + p[1])
(0, 10)
```

Because an anonymous function is an expression, it can also be `return`ed (and `yield`ed) by other functions. For instance, suppose we want to generalize the concept of taking the minimum of a list of tuples by comparing the kth element of each tuple. For pairs, like above, we would use $k = 0$ for the first component and $k = 1$ for the second component. In general we can compute a kth-projection function:

```
1 def project(k):
2     """Returns a function that would pick the kth component
3     of its input"""
4     return lambda seq: seq[k]
```

Now we can recast our `min()` example above:

```
>>> min(l, key=project(0))     # element with smallest first component
(-10, 157)
>>> min(l, key=project(1))     # element with smallest second component
(0, 10)
```

A TOOLBOX FOR ANONYMOUS FUNCTIONS There are two built-in functions in Python that implement classic use cases for anonymous functions: applying a function to all elements (`map`), and selecting elements that satisfy a condition (`filter`). You would see these functions in other functional programming languages such as Haskell and OCaml, and they are also often found in libraries. In addition, there are a number of other useful tools in the `functools` and `itertools` modules that are well suited for anonymous functions. Here is a small selection.

- `map(func, iterable)`: this function runs the given argument `func` on each element of the iterable. The result of the `map()` call is itself an iterable that returns the result of the pointwise application of `func`. Here, for instance, is an iterable that doubles every element of a list.

  ```
  >>> map(lambda x: x * 2, [0, -10, 42, 157])
  <map object at 0x7f81696e8cc0>
  >>> list(map(lambda x: x * 2, [0, -10, 42, 157]))
  [0, -20, 84, 314]
  ```

- `filter(func, iterable)`: this is like `map()`, but it instead runs `func` on each element of the `iterable` to determine whether it belongs in the output or not. If `func` returns `True`, the element is kept; otherwise, it is discarded. An example:

  ```
  >>> list(filter(lambda x: (x % 2 is 0), [0, -10, 42, 157])) # select even numbers
  [0, -10, 42]
  ```

- `reduce(func, iterable)`: if `iterable` consists of at least two elements, then this returns the result of chained applications of `func` to these elements. For instance:

  ```
  >>> from functools import reduce
  >>> reduce(lambda x, y: x + y, [1, 2, 3])         # compute the sum
  6
  >>> reduce(lambda x, y: max(x, y), [1, 2, 3])     # compute the max
  3
  >>> reduce(max, [1, 2, 3])                        # identical to above
  3
  ```

  Observe that we are not required to use anonymous functions here: for a function like `max()`, writing `lambda x, y: max(x, y)` is basically the same as writing just `max` if we know that it will only be applied to two arguments.

  In some other functional programming languages, `reduce()` is often called `fold()`.

ANONYMOUS FUNCTIONS VS. COMPREHENSIONS As should be obvious, there is some overlap in functionality between anonymous functions and comprehension expressions when it comes to computing with iterables. In fact, `map()` and `filter()` are already built into the syntax of comprehensions, so we can see the following equivalences:

```
list(map(f, itbl))        [f(x) for x in itbl]
list(filter(p, itbl))     [x for x in itbl if p(x)]
```

In fact we can combine both `map` and `filter` functionality into a comprehension at once:

```
[f(x) for x in itbl if p(x)]
```

On the other hand, these comprehensions cannot easily mimic the functionality of `reduce()`. Whether you use these functions from the functional programming toolbox or use the built in comprehensions is often a matter of taste, but you need to know both styles if you want to read code from other people.

1.4.1 *Exercises*

1. *(Fizz Buzz)* Using anonymous functions, fill in the blank in the following function that converts an iterable `itbl` for natural numbers to an iterable that replaces every multiple of `3` with `'Fizz'`, every multiple of `5` with `'Buzz'`, multiples of *both* `3` and `5` with `'FizzBuzz'`, and leaves all other numbers unchanged.

```
def fizz_buzz(itbl):
    return map(_____________, itbl)
```

```
>>> list(fizz_buzz(range(20)))
['FizzBuzz', 1, 2, 'Fizz', 4, 'Buzz', 'Fizz', 7, 8, 'Fizz',
 'Buzz', 11, 'Fizz', 13, 14, 'FizzBuzz', 16, 17, 'Fizz', 19]
```

   Hint: use the ternary conditional expression `(res1 if cond else res2)` that computes `res1` if `cond` is `True` and `res2` otherwise.

2. Using anonymous functions fill in the blank in the following function that implements set intersection for the two input sets `s1` and `s2`, i.e., it computes a new set that contains (only) every element that is in both `s1` and `s2`.

```
def set_intersection(s1, s2):
    return set(filter(_____________, s1))
```

3. The `reduce()` function takes an optional third argument that is the initial value of the accumulator that aggregates the result of applying the function. That is, `reduce(f, itbl, initial)` computes

   $f(f(\ldots f(f(\text{initial}, x_1), x_2), \ldots), x_n)$

   where $x_1, x_2, \ldots, x_n$ are the elements in `itbl`. Using anonymous functions, fill in the blank in following function that turns a list of decimal digits into the corresponding number.

```
def to_number(digits):
    return reduce(_____________, digits, _____________)
```

```
>>> to_number([4, 2])
42
>>> to_number([])
0
```

1.5 SUBCLASSING AND INHERITANCE

Python's `class` construct supports some aspects of *object-oriented programming* (OOP). There are two conceptual categories involved in Python's variant of OOP: *objects* and *classes*. A class is defined using the `class` keyword and is a kind of *blueprint* to build *instances* of the class. For instance, take this simple class:

```
class A:
    def __init__(self):
        self.x = 42

    def func(self, y):
        return self.x + y
```

Classes such as these can be used to stamp out *instance objects*; for instance, to create an instance a of the class `A`, we would say:

```
>>> a = A()
```

Such an instance can be seen as being built by starting with an "empty" object and then running the class initializer (the `.__init__()` special function). In other words, the above would be equivalent to the following, were it to be legal Python.

```
>>> a = object()
>>> A.__init__(a)
```

Calls to member functions on an object instance are similarly transformed to calls to the corresponding function in the class, where the `self` special argument is replaced by the object instance. Therefore, objects are not classes but are rather instances of classes, and classes are not objects themselves but rather contain the definitions of the associated functionality of their instance objects.[7]

SUBCLASSING A class `B` can be *derived* from the class `A` which marks it as a *subclass*. To mark a class as a subclass, add the parent class(es) in parentheses in the `class` declaration. For example:

```
class B(A):
    pass
```

The effect of this declaration is, effectively, to import the definition of the class `A` into `B`, which means that it *inherits* `A`'s `.__init__()` and `.func()` member functions. In addition to this inheritance, this declaration also extends the `isinstance()` relation, so that any instance of `B` is also considered an instance of `A`.

```
>>> b = B()
>>> isinstance(b, B)
True
>>> isinstance(b, A)
True
```

Like the `isinstance()` built-in function, there is also an `issubclass()` built-in function that can be used to check that one class is an instance of another.

[7] Strictly speaking, classes in Python can be interpreted as a kind of object. Indeed, Python can even create brand new classes at run time, which leads to self-modifying code. This kind of *meta-programming* is beyond the scope of this course.

```
>>> issubclass(A, B)
False
>>> issubclass(B, A)
True
```

Overriding definitions Subclasses are allowed to *override* the definitions of member functions by providing their own. The member function of the parent class is effectively suppressed or hidden, and replaced with that of the subclass. For example:

```
1  class A:
2      def func(self):
3          print('A')
4
5  class B(A):
6      def func(self):
7          print('B')
```

```
>>> a = A()
>>> a.func()
A
>>> b = B()
>>> b.func()
B
```

Overriding is particularly useful when the subclass redefines a member function that is used by the parent class. The functionality of the member function of the parent class changes to reflect the new definitions in the subclass. Here is an evocative example.

```
1  class Animal:
2      def noise(self):
3          return 'generic animal noise'
4
5      def describe(self):
6          print('Noise:', self.noise())
7
8  class Dog(Animal):
9      def noise(self): # override definition in class Animal
10         return 'woof!'
```

```
>>> a = Animal()
>>> a.describe()
Noise: generic animal noise
>>> d = Dog()
>>> d.describe()
Noise: woof!
```

In other words, even though in line 6 the call to `.noise()` is part of a definition in the class `Animal`, the fact that this function is overridden in class `Dog` ensures that it is `Dog`'s definition that is actually used.

Inside of a subclass, you can still access the definitions of the parent class using the built-in **`super()`** function. The precise details of how **`super()`** works are too complicated and Python-specific, so it is enough to see it as a way of accessing the original definitions from overridden definitions. If a class has multiple parents, then **`super()`** works in an intelligent way to pick the first class in the list of parents that has a compatible definition.

```
class Dog(Animal):
    def noise(self):
        return 'woof! (instead of {})'.format(super().noise())
```

```
>>> d = Dog()
>>> d.describe()
Noise: woof! (instead of generic animal noise)
```

1.6 KEYWORD, OPTIONAL, AND VARIABLE ARGUMENTS

Required arguments Consider a Python function such as the following.

```
def func(m, n):
    print('m = {}; n = {}'.format(m, n))
```

Such a function can be *called* in a number of ways.

- *Positional arguments*: here the first argument is bound to `m`, the second to `n`, and so on. The position of the argument in the list of arguments determines which variable in the definition of the function is bound to it.

  ```
  >>> func(10, 20)
  m = 10, n = 20
  >>> func(20, 10)
  m = 20, n = 10
  ```

- *Keyword arguments* (aka *named arguments*): here the variable in the definition of the function is explicitly mapped to the argument it is bound to using `=`. The order in which these arguments are written does not matter.

  ```
  >>> func(m=10, n=20)
  m = 10, n = 20
  >>> func(n=20, m=10)
  m = 10, n = 20
  ```

- The two styles can be combined, but positional arguments must come first.

  ```
  >>> func(10, n=20)
  m = 10, n = 20
  >>> func(m=10, 20)
    File "<stdin>", line 1
  SyntaxError: positional argument follows keyword argument
  ```

  Moreover, each variable must be unambiguously bound with either a positional argument or a keyword argument, not both.

  ```
  >>> func(10, m=20)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: func() got multiple values for argument 'm'
  ```

Optional arguments Sometimes it is useful to define a function with *optional arguments* that may be omitted when calling the function; to specify an argument as optional, a default value must be provided in the definition of the function.

```
def func(m, n, k=30):
    print('m = {}, n = {}, k = {}'.format(m, n, k))
```

Such optional arguments, when provided, can be provided either positionally or as keyword arguments, with the same restrictions as for required arguments.

```
>>> func(10, 20)              # omitted; default plugged in
m = 10, n = 20, k = 30
>>> func(10, 20, 42)          # positional
m = 10, n = 20, k = 42
>>> func(10, 20, k=42)        # keyword
m = 10, n = 20, k = 42
>>> func(10, k=42, n=20)      # order of keyword arguments doesn't matter
m = 10, n = 20, k = 42
```

Using default values for arguments comes with a few caveats. First, the expression that is used to compute the default value of an argument must be computable *before* the definition of the function itself. In particular, the default argument cannot depend on the values of any of the other arguments, so the following is not legal.

```
def func(x, y, z=x + y):
    # ...
```

This kind of situation must be handled by setting the default value to `None` and then adding a line of code to the body of the function to compute a default value in case none was supplied.

```
def func(x, y, z=None):
    if z is None: z = x + y
    # ...
```

The second caveat is that the default value of an argument is computed *once* when the function is defined, instead of repeatedly being recomputed. As a consequence, unexpected things will happen if the default value is changed in the call to the function. For instance, it is a common pattern to pass a book-keeping object around during recursive calls. Consider a function that does a depth-first exploration of a graph (covered in more detail in Chapters 4 and 5):

```
def dfs(graph, start_node, seen_nodes=set()):
    # ... eventually:
    seen_nodes.add(node)
    # recursive call passes seen_nodes around
    dfs(graph, node, seen_nodes)
    # ...
```

In code of this form, after the first call to `dfs(graph, node`$_1$`)` completes, the next time `dfs(graph, node`$_2$`)` is called the `seen_nodes` set still contains the nodes added in the first call, instead of being the empty set as expected. This will make the second call to `dfs()` return incorrect results, since too many nodes will be in the `seen_nodes` set. To avoid this pitfall, default values that change must be handled in the same way as default values that cannot be computed early, using the `None` trick.

```
def dfs(graph, start_node, seen_nodes=None):
    if seen_nodes is None: seen_nodes = set()
    ## rest of the function as before
```

VARIABLE ARGUMENTS Python also allows defining functions that take an arbitrary number of arguments. Some examples of functions that you have already seen with such features include:

- The `print()` function that takes an arbitrary number of positional arguments and prints them one by one, separated by spaces (by default).

```
>>> print(10, 3.14, 'hello')
10 3.14 hello
```

- The `dict()` function that takes an arbitrary number of keyword arguments and builds a dictionary from them with the name of the keyword argument turned into the key of the dictionary.

```
>>> dict(m=10, n=20, k='hello')
{'m': 10, 'n': 20, 'k': 'hello'}
```

To define functions that support such arguments, we have to use two special forms for arguments.

- Arbitrary positional arguments are specified using an argument name that begins with `*`, such as `*args`. This stands for the tuple of all the positional arguments that are not already mapped to other existing arguments. For instance, here is how we can write a function `multiply()` that returns the product of its positional arguments.

```
1  def multiply(*args):
2      product = 1
3      for n in args: product *= n
4      return product
```

  Note that the argument tuple is named `args`, not `*args`; the latter is only used when declaring the argument (in the `def` statement) to mark it as the tuple holding all the remaining positional arguments.

```
>>> multiply(10, 20, 30)
6000
>>> multiply()
1
```

- Arbitrary keyword arguments are specified using `**` in front of the argument name, which is called `kwargs` by convention. This argument itself is a dictionary mapping the keywords to the arguments they are bound to. For example, here is a function that takes an arbitrary number of keyword arguments bound to integers, and splits them into two lists of keywords bound to even numbers and odd numbers respectively.

```
1  def split_keywords(**kwargs):
2      evens, odds = [], []
3      for k, v in kwargs.items():
4          if v % 2 == 0: evens.append(k)
5          else: odds.append(k)
6      return (evens, odds)
```

```
>>> split_keywords(a=1, b=1, c=2, d=3, e=5, f=8)
(['c', 'f'], ['a', 'b', 'd', 'e'])
```

1.7 REGULAR EXPRESSIONS

PURPOSE AND USAGE *Regular expressions* are a mini language for describing patterns of substrings in a string. Each regular expression stands for a set of strings that *match* the expression, which is sometimes called the *language* of the expression. The term *regular*, in fact, comes from the mathematical study of formal languages that you will see in later courses. Regular expressions are built into Python and their functionality is sequestered to the `re` library.

```
1 import re
```

Regular expression objects are *compiled* from ordinary Python strings, but certain characters in the string have special meanings. We will explain the syntax for regular expressions below, but let us first see how to use them in Python. We will use the example of `'m(iss)+ippi'`, which stands for the regular expression that matches the set of strings {`'missippi'`, `'mississippi'`, `'mississississippi'`, ...}.

To create a regular expression object, also known as a *pattern*, we would take this string that describes the expression and use the function `re.compile()`.

```
>>> ex = re.compile('m(iss)+ippi')
```

This creates an instance of the `re.Pattern` class.

```
>>> isinstance(ex, re.Pattern)
True
```

To use a pattern to match a string, we use the `.fullmatch()` member function of the pattern object.

```
>>> ex.fullmatch('mississippi')
<re.Match object; span=(0, 11), match='mississippi'>
>>> ex.fullmatch('mippi')
# returns None
```

When the match fails, it returns `None`, and when it succeeds, it returns an instance of `re.Match`. Every instance of `re.Match` implicitly converts to the Boolean `True`, so `.fullmatch()` can be used in an `if ... else ...` expression.

```
>>> if re.fullmatch('mississippi'): print('matched!')
...
matched!
```

Another important member function of patterns is `.search()`, that can be used to find a substring that matches the pattern. For example:

```
>>> ex.search('one mississippi, two mississippi')
<re.Match object; span=(4, 15), match='mississippi'>
```

This also returns a `re.Match` instance, but in this case the substring that is matched is not at the start of the input string. Indeed, the matched string begins at position 4 (the position of the `'m'`), and ends at position 14 (the position of the `'i'`), meaning that the range of characters positions in the string is `range(4, 15)`. This is called the *span* of the match, and can be extracted from the match object using the `.span()` function.

```
>>> ex.search('one mississippi, two mississippi').span()
(4, 15)
```

Note that there is more than one instance of the pattern in the string. If we wanted to iterate over all the (non-overlapping!) substrings of the input string that match the pattern, we would use the `.finditer()` member function.

```
>>> for mat in ex.finditer('one missippi, two mississippi, three mississsissippi'):
...     print(mat.span())
...
(4, 12)
(17, 28)
(36, 50)
```

Syntax of regular expressions Since we don't have direct access to the pattern objects, we will always talk about the strings that get compiled to patterns using `re.compile()`. We will call such strings *regular expressions* as well, which is common practice if slightly misleading.

- *Concatenation*: if R_1 and R_2 are regular expressions, then so is R_1R_2, and it matches strings that are the concatenation of two strings where the left component matches R_1 and the right component matches R_2.
- *Alphanumeric characters*: every letter or number stands for itself, i.e., matches exactly that character. Thus, a concatenation of alphanumeric characters only matches the same string. For example, `2fast2furious` matches the singleton set of strings {`'2fast2furious'`}.
- *Set of characters*: a list of characters delimited by `[` and `]` defines a regular expression that matches a single character belonging to the list. For example, `[abcd]` matches the set {`'a'`, `'b'`, `'c'`, `'d'`}. To specify a range of characters, the start and (inclusive) end of the range can be joined with `-`. Thus, `[a-z]` matches the set {`'a'`, `'b'`, ..., `'z'`}.
- *Complementary set of characters*: the dual of the previous construct, a list of characters delimited by `[^` and `]` matches any character *except* the ones belonging to the list. E.g., `[^a-z]` would match any character except those in the set {`'a'`, `'b'`, ..., `'z'`}.
- *Any arbitrary character*: the regular expression `.` matches any single arbitrary character.
- *Grouping*: if R is a regular expression, then so is `(`R`)` and it matches the same set of strings. The purpose of the parentheses is to group the components of R together so one of the operators below can be applied to the entire expression. These parentheses also serve to identify groups and for extracting sub-matches, which we will describe below.
- *Alternation*: if R_1 and R_2 are regular expressions, then so is R_1`|`R_2 and it matches a string that either matches R_1 or, if R_1 does not match, then matches R_2. E.g., `abc|xyz` matches {`'abc'`, `'xyz'`}.
- *Repetition*: if R is a regular expression, then R`*` is a regular expression that stands for 0 or more repeated matches of R. Thus `(iss)*` matches the set {`''`, `'iss'`, `'ississ'`, ...}.
- *Non-empty repetition*: if R is a regular expression, then R+ is a regular expression that stands for 1 or more repeated matches of R. In other words, it is the same as RR`*`.
- *Numbered repetition*: for more fine-gained control over repetition, we have two numbered forms. For a regular expression R, we have:
  - R`{m}` (where `m` is a literal decimal integer) to indicate that R is to be matched exactly `m` times. Thus `a{3}` matches {`'aaa'`}.
  - R`{m,n}` (where `m` and `n` re literal decimal numbers) to indicate that R is to be matched at least `m` times *and* no more than `n` times. E.g., `a{3,5}` matches {`'aaa'`, `'aaaa'`, `'aaaaa'`}. Both `m` and `n` can be omitted to indicate that there is no corresponding minimum or maximum. If

both are omitted, then `R{,}` is a synonym for `R*`. E.g., `a{3,}` matches {`'aaa'`, `'aaaa'`, ...} and `a{,5}` matches {`''`, `'a'`, `'aa'`, `'aaa'`, `'aaaa'`, `'aaaaa'`}

- *Optional*: if `R` is a regular expression, then so is `R?`, and it matches a string that matches *R* if possible, and the empty string of a match of *R* is not possible.
- *Escapes*: if we need to match the literal characters `*`, `+`, `?`, etc., then they have to be prefixed with a backslash in the pattern. That is, the regular expression that matches the string `'*'` would be `\*`. Note that since regular expressions are represented as strings in Python, and backslashes have a special meaning in ordinary Python strings, this poses a problem: to write the literal string `\*` we would have to use `'\\*'`. To avoid this complication, we can add an `r` in front of the string to mark it as a *raw string*, i.e., we can write `r'\*'` to indicate that backslashes should not be interpreted. To match the backslash character itself, we would use two backslashes: `r'\\'`.

In addition to these basic components, regular expressions also have a few *meta-operators*.

- *Beginning of Input*: the regular expression `\A` matches the empty string but only at beginning of the input string. If any of the characters of the string have already been matched, then this regular expression will fail to match any string.
- *Beginning of Line*: the regular expression `^` matches the empty string at every beginning of a line. This differs from `\A` only if the input string spans several lines, and the regular expression matching is in a so-called *multiline* mode. In the default mode (single line), the two regular expressions `^` and `\A` coincide.
- *End of Input*: dually, the regular expression `\Z` matches the empty string at the end of the input string, and fails to match anything at any other position.
- *End of Line*: the regular expression `$` matches the empty string at every end of a line. This differs from `\Z` only if the input string spans several lines, and the regular expression matching is in *multiline* mode. In the default single line mode, the two regular expressions `$` and `\Z` coincide.
- *Digits and non-digits*: the regular expression `\d` matches any decimal digit. It includes the set `[0-9]` but may in addition include decimal digits in other writing systems (e.g., Arabic). The complementary operator, `\D`, matches any character that is *not* a decimal digit.
- *Whitespace and non-whitespace*: the regular expression `\s` matches any whitespace character, which includes the space, tab, line- feed, newline, and vertical tab (rare) characters. It may also include space characters in other writing systems, such as the *double-width space* character used in Chinese or the *non-breaking space* character used in digital type-setting. Its complementary operator is `\S`, which matches any character that is not a whitespace character.
- *Word and non-word*: the regular expression `\w` matches any character that may be used in a *word*, understood broadly. For Latin, this includes the set `[a-zA-Z0-9_]`, but does not include punctuation, whitespace, etc. The complement, `\W`, matches any character that is not a word character.

MATCHING GROUPS As mentioned above, parentheses are used to group parts of a regular expression together so regular expression operators such as `*` and `?` can be applied to more than basic regular expressions. In particular, the expression `ab?` is identical to `a(b?)`, not `(ab)?`. Interestingly, this is not the case for the alternation operator `|`, which has the lowest possible precedence; thus, `ab|yz` is identical to `(ab)|(yz)` and not `a(b|y)z`.

It turns out that parentheses are also useful for extracting *parts* of a matched string. Each opening parenthesis `(` in scanning the regular expression from left to right defines a *matching group*, and each such group ends with its corresponding matching closing parenthesis `)`. The entire regular expression is given

group number 0, and the other groups are numbered in sequence starting from 1. For example, in the regular expression:

```
m(i(s+))+((ip*)*i)
```

there are the following matching groups:

group	regular expression
0	`m(i(s+))+((ip*)*i)`
1	`i(s+)`
2	`s+`
3	`(ip*)*i`
4	`ip*`

Note that operators such as `*` or `?` that are applied *to* a matching group are not considered to be part of the group. A matching group consists only of the portion of the regular expression between a matching pair of parentheses.

To extract the part of the matched input string that corresponds to a given matching group number, we would use the `.group()` function of the `re.Match` object.

```
>>> ex = re.compile(r'm(i(s+))+((ip*)*i)')
>>> mat = ex.fullmatch('mississippi')
>>> mat.group(0)
'mississippi'
>>> mat.group(1)
'iss'
>>> [mat.group(i) for i in range(5)]
['mississippi', 'iss', 'ss', 'ippi', 'ipp']
>>> mat.group(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
```

The spans of the matches corresponding to the matching groups can be obtained using the `.span()` function, where the group number can be specified in the optional argument (which defaults to 0, i.e., the entire regular expression).

```
>>> [mat.span(i) for i in range(5)]
[(0, 11), (4, 7), (5, 7), (7, 11), (7, 10)]
```

REPLACEMENT An important use for regular expressions is to *replace* the matched portion of an input string with a different string. For example, we may want to replace every occurrence of the substring `'s'` with the string `'p'` in the input string, which would change `'mississippi'` to `'mippippippi'`. To perform the replacement we would use the `.subn()` function of the regular expression object.

```
>>> ex = re.compile(r's')
>>> ex.subn('p', 'mississippi')
('mippippippi', 4)
```

Note that the `.subn()` function takes the replacement string as its *first* argument, and the input string as its second argument. It also returns a tuple of the resulting string and a count of the number of replacements made. The `.subn()` function also takes an optional argument `count` that places an upper bound on the number of replacements

```
>>> ex.subn('p', 'mississippi', count=2)
('mippissippi', 2)
```

Replacements can be made for any regular expression and the replacement strings can be of any size.

```
>>> ex = re.compile('is+')
>>> ex.subn('ith', 'mississippi')
('mithithippi', 2)
>>> ex.subn('iz', 'mississippi')
('mizizippi', 2)
```

Referring to matching groups In the replacement text, it is sometimes useful to refer to the matched groups in the text. For instance, suppose we wanted to change the order of the `iss` and `ipp` substrings of the string `'mississippi'`. We could use the regular expression `m((is+)*)((ip+)*)i` so that the matching groups 1 and 3 contain the substrings with `s` and `p` respectively.

```
>>> ex = re.compile(r'm((is+)*)((ip+)*)i')
>>> mat = ex.fullmatch('mississippippippi')
>>> mat.group(1)
'ississ'
>>> mat.group(3)
'ippippipp'
```

To refer to these matched substrings in the replacement text, we would use *numbered references* of the form `\1`, `\2`, etc. to stand for the matched group 1, 2, etc. (respectively). Note that these references occur in the *replacement text*, not in the input string nor in the regular expression. Thus, to swap the groups 1 and 3 in the input text above, we would use the replacement string `r'm\3\1i'`.

```
>>> ex.subn(r'm\3\1i', 'mississippippippi')
('mippippippississi', 1)
```

2 BINARY REPRESENTATION

Modern digital computers are built to work with *digital signals* that have a discrete discontinuous range, such as $\{0V, +5V\}$, or, more abstractly, {on, off}. These discrete values can not only be used during computation but also be *recorded* and *transmitted* as streams of on/off values – commonly represented with the literal 0 (for off) and 1 (for on). At a fundamental level, therefore, all digital data is just a stream of 0s and 1s. A computer does not inherently know what an integer, a string, a function, a file, a program, a network, etc. are; rather, *we* as programmers and designers must decide how to represent all such concepts by *encoding* or *representing* them as sequences of 0s and 1s. These encodings change all the time as new forms of computation are invented and new uses for computing are found, and as a consequence a computer from even a decade ago will often have a hard time processing new data from today, even if they are able to read sequences of 0s and 1s just fine.

If we want to understand—or design—a digital encoding for some concept, we have to be able to program at the low representational level of streams of 0s and 1*s*. This chapter will give you an overview of this kind of low level programming.

2.1 BITS AND BYTES

The term *bit* was supposedly coined by Claude Shannon[1] as a contraction of *binary digit*. Thus, a bit is either a 0 or a 1. Bits are almost never encountered in isolation; rather, 8 bits are packed into a *byte*, which is the smallest unit of storage. There are 256 possible *values* that a byte can have (since $2^8 = 256$). Here they are:

```
00000000  00000001  00000010  ···  11111110  11111111
```

Note that the order of the bits matters: the byte `00101010` is different from `10101000` even though they have the same number of `1`s and `0`s.

The nominal value of a byte is the natural number that is represented by the standard left-to-right positional interpretation as a binary number, just like in the decimal representation of natural numbers. That is, if the byte consists of the bits $b_0, b_1, \ldots, b_7$ laid out like so:

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

then its nominal value is:

$$2^0 b_0 + 2^1 b_1 + 2^2 b_2 + 2^3 b_3 + 2^4 b_4 + 2^5 b_5 + 2^6 b_6 + 2^7 b_7.$$

Thus `00101010` has the nominal value 42 and `10101000` has the nominal value 168.

The bit that is written first in the left-to-right positional representation is multiplied by the largest power of 2, and is therefore called the *most significant bit* (MSB) and likewise the rightmost bit is called the *least significant bit* (LSB). By convention bit positions are always numbered from the least to the most significant bit, so when we say the *k*th bit, we mean the *k*th least significant bit numbering from 0.

For numbers larger than 256, we will need more than a single byte's worth of bits. A modern processor actually computes with *words* that are multiple bytes long: 4 bytes (32 bits), 8 bytes (64 bits), and in some rare instances even 16 bytes (128 bits). Processors with 1 byte (8 bit) and 2 byte (16 bit) words have also

[1] Remember that name, because we will meet him again in Section 7.3.

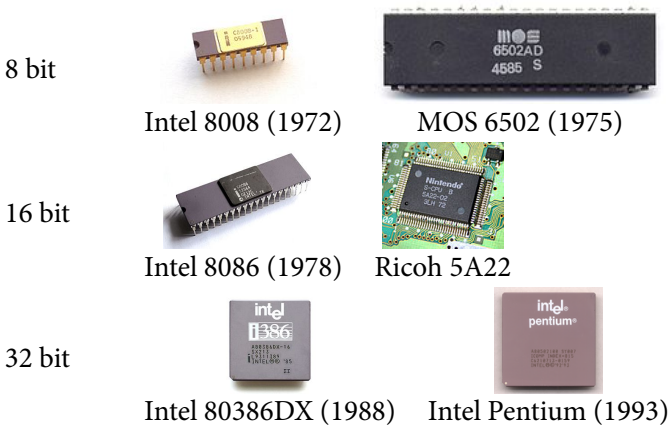


Figure 2.1: Some famous processors and their associated word sizes

existed in the past, but they are vanishingly rare now in general purpose computing hardware. Such small word sizes are still used in *microcontrollers*, which are special purpose processors to control hardware devices such as robots or drones. The number of bits in a word determines the number of parallel lanes of traffic there must be between the various components of the computer—register banks, the arithmetic and logic units, IO devices, main memory, etc.—that minimizes communication time, since each bit of the word can be transferred in parallel to all the other bits. Figure 2.1 shows some images of famous processors and their associated word sizes.

Larger collections of bytes are measured in one of two scales, depending on whether your base of exponentiation is 10 (like in the natural sciences) or 2 (as is more common in computer science). The base 10 scale, sometimes called the *metric scale*, should be familiar: 1000 bytes makes a *kilobyte*, 1000 kilobytes a *megabyte*, and so on. The base 2 scale, sometimes called the *binary scale*, uses almost the same prefixes, but it replaces the final syllable with *bi*; thus, *kilo* becomes *kibi*, *mega* becomes *mebi*, etc. After prefixes like *kilo* or *kibi*, we say *bit* or *byte* if we are counting bits or bytes. Here are some of the points in two scales together with their name and abbreviation.

10^3 bits	*kilobit*	kb	2^{10} bits	*kibibit*	Kib
10^3 bytes	*kilobyte*	kB	2^{10} bytes	*kibibyte*	KiB
10^6 bits	*megabit*	Mb	2^{20} bits	*mebibit*	Mib
10^6 bytes	*megabyte*	MB	2^{20} bytes	*mebibyte*	MiB
10^9 bits	*gigabit*	Gb	2^{30} bits	*gibibit*	Gib
10^9 bytes	*gigabyte*	GB	2^{30} bytes	*gibibyte*	GiB
10^{12} bits	*terabit*	Tb	2^{40} bits	*tebibit*	Tib
10^{12} bytes	*terabyte*	TB	2^{40} bytes	*tebibyte*	TiB

It is important to pay attention to these abbreviations since different scales and units are used in different contexts. For example, hard drives are generally always advertised using the metric scale, so when you by a 10 gigabyte thumb drive you may be surprised to find that it has only 9.54 GiB of space. Similarly, if you see a connection speed advertised as 100 Mbps, then you will only be able to send $100/8 = 12.5$ million bytes every second.

2.2 ACCESSING THE BIT REPRESENTATION

You already know that there are functions like `int()`, `float()`, etc. to convert arbitrary values to `int`, `float`, etc. Another one of these primitive functions is `bin()` that produces a Python string—i.e., a `str` value—made up of the characters `'0'` and `'1'`, which contains the bit representation of the argument of the function. This `str` is always prefixed by the string `'0b'` to identify it as a *bit string*. Some examples:

```
>>> bin(0)
'0b0'
>>> bin(42)
'0b101010'
>>> bin(128)
'0b100000000'
>>> bin(1048575)
'0b11111111111111111111'
```

Why `'0b'`? Because in Python you can write literal numbers in binary notation by using the prefix `0b` (note: no quotes). Ordinary literal numbers that occur in your code are in decimal notation, but Python also supports three other bases: binary, octal, and hexadecimal. In all cases, however, the values being defined are `int` values.

```
>>> 42          # by default literals are in decimal
42
>>> 0b101010    # the prefix 0b stands for binary
42
>>> 0o52        # the prefix 0o stands for octal (base 8, digits 0-7)
42
>>> 0x2a        # the prefix 0x stands for hexadecimal (base 16, digits 0-9,a-f)
42
>>> 42 == 0b101010 == 0o52 == 0x2a
True
```

Keep in mind that a numerical literal and a string are *not* the same in Python.

```
>>> 0b101010 == '0b101010'
False
```

However, you can convert the string `'0b101010'` back to the int `42` using the `int()` function, although you have to give an explicit base argument since the default is base 10.

```
>>> int('0b101010', 2)
42
```

2.3 TWO'S COMPLEMENT REPRESENTATION

Since a bit can only be a 0 or a 1, the nominal value of sequence of bits can only be a non-negative integer. If we want to represent negative integers, we have to *encode* it somehow as a sequence of bits, which amounts to declaring certain nominal values to *stand for* negative integers. This can seem paradoxical. But, remember that it is OK if the computer has no inherent knowledge of what we are representing with bit sequences. It is enough that the bit sequences in the range of the encoding are in one-to-one correspondence with the things being encoded, and that operations on the things being encoded are preserved by the encoding. Let us see this abstract notion in action for representing negative integers.

The representation technique that is used in practically every computing system that exists today is known as *two's complement representation*. In order to define it, we must first fix a given word size of W

bits. (As you are reading this chapter, check your intuitions with $W = 8$ and $W = 16$.) We have 2^W possible things that we can represent with W bits.

Definition 2.1. *The* **two's complement** *of a number x* with respect to *a word size W is the number $2^W - x$. This complement is written as $\overline{x}^W$.*

Directly from its definition we have that $\overline{\overline{x}^W}^W = x$. Moreover, $x + \overline{x}^W = 2^W \equiv 0 \pmod{2^W}$. This suggests a simple representation technique for integers in the range $\{-2^{W-1}, \ldots, -1, 0, 1, \ldots, 2^{W-1}-1\}$, i.e., the range $[-2^{W-1}, 2^{W-1})$. Note that this is just the range $[0, 2^W)$ shifted to the left by 2^{W-1}.

Definition 2.2. *The* **two's complement representation** *of a (signed) integer x using W bits, written as* $\mathsf{twoc}(x, W)$*, is a non-negative integer defined as follows:*

- *If $x \geq 0$, then* $\mathsf{twoc}(x, W)$ *has the same bits as x*
- *If $x < 0$, then* $\mathsf{twoc}(x, W)$ *uses the bits of $\overline{-x}^W$.*

This representation is only defined if $x \in [-2^{W-1}, 2^{W-1})$.

It is important to note that no matter what x is, its representation $\mathsf{twoc}(x, W)$ is a positive integer. It's just that some of these positive values of $\mathsf{twoc}(x, W)$ are *interpreted* to mean negative integers. This interpretation is entirely on the part of the programming language and the programmer. Your processor knows nothing about signed integers: all its arithmetic operations are defined with the assumption that integers are non-negative.

Here are a few illustrative examples of this representation with $W = 8$:

x	$\mathsf{twoc}(x, 8)$
0	`0b00000000`
42	`0b00101010`
127	`0b01111111`

x	$\mathsf{twoc}(x, 8)$
-1	`0b11111111`
-42	`0b11010110`
-128	`0b10000000`

It should be immediately apparent that if the most significant bit of the representation is a `1`, then the number being represented is negative, and otherwise it is positive. Hence, this bit is often known as a *sign bit*. Another interesting fact is that both addition and subtraction can be implemented using just addition.

Theorem 2.3 (Isomorphism)**.** *The following are true whenever the representations are well-defined.*

1. $\mathsf{twoc}(x + y, W) = \mathsf{twoc}(x, W) + \mathsf{twoc}(y, W)$
2. $\mathsf{twoc}(x - y, W) = \mathsf{twoc}(x, W) + \mathsf{twoc}(-y, W)$
3. $\mathsf{twoc}(x \times y, W) = \mathsf{twoc}(x, W) * \mathsf{twoc}(y, W)$

Proof. See Exercise 1. □

To summarize, two's complement representations allow us to represent integers between -2^{W-1} and $2^{w-1} - 1$ (ends inclusive) using words of size W. Thus, with a word size of 32 bits, we can represent integers between -2^{31} and $2^{31} - 1$. Using the rule of thumb that 2^{10k} is approximately 10^{3k}, we see that the absolute range—i.e., the range of absolute values—of signed 32 bit integers is about 2 billion. With 64 bits, the absolute range is approximately 9 quintillion.

2.4 BITWISE OPERATIONS

SHIFTING Let us now turn to how we may programmatically test and modify any arbitrary bit in the two's complement representation of integers with a word size of W. In order to do this, it is very useful to

have access to those numbers whose bit patterns are all `0`s except for a single occurrence of a `1`. Of course such numbers are powers of 2, so the first thing to find out is how to compute the kth power of 2.

You already know about the exponentiation operator `**`, so one simple way to get this 2^k is to say `2 ** k`. However, this answer is Python-specific as integer exponentiation is not a primitive operation in your processor. It is implemented in software in terms of a more primitive operation known as *shifting*. There are two such operations, *left shift* and *right shift*, which are roughly dual to each other.

Definition 2.4 (bit shifting). *Let a word size W and a non-negative number $k \leq W$ be given.*

Left shift *Given a W-bit word w with, its k-left-shifted version is written w<< k, which is a W-bit word with the following definition.*

$$\texttt{0b}b_{W-1}\cdots b_{W-k}b_{W-k-1}\cdots b_0 \texttt{ << } k \quad = \quad \texttt{0b}b_{W-k-1}\cdots b_0 \underbrace{\texttt{00}\cdots\texttt{0}}_{k \text{ bits}}$$

In other words, the most significant k bits are lopped off and k `0`s are added to the right.

Right shift *Given a W-bit word w, its k-right-shifted version is written w >> k, which is a W-bit word with the following definition:*

$$\texttt{0b}b_{W-1}\cdots b_k b_{k-1}\cdots b_0 \texttt{ >> } k \quad = \quad \texttt{0b}\underbrace{b_{W-1}b_{W-1}\cdots b_{W-1}}_{k \text{ copies of } b_{W-1}} b_{W-1}\cdots b_k$$

In other words, the least significant k bits are looped off and k copies of the sign bit, b_{W-1}, are added to the left.

Note that if the word size W is unbounded, as is the case in Python (but not in hardware), then there is no need to lop off the `k` most significant bits when shifting left by `k`. Shifting right always removes the `k` least significant bits, however.

Shifting a number left by k turns out to be the same as multiplying by $2^k \pmod{2^W}$. (Verify it!) For right shifts, you may wonder why the sign bit (i.e., the MSB) is copied instead of filling with `0`s like with left shifts. This is because the intuition is that a right shift is the opposite of a left shift, i.e., shifting a non-negative number right by k should correspond to dividing (using integer division, `//`) the number by 2^k. It's easy to see that this is indeed the case for non-negative integers:

$$\left(\sum_{i=0}^{W-1} 2^i b_i\right) \texttt{ // } 2^k \;=\; \left\lfloor \frac{1}{2^k}\left(\sum_{i=0}^{W-1} 2^i b_i\right)\right\rfloor \;=\; \left\lfloor \sum_{i=0}^{W-1} 2^{i-k} b_i \right\rfloor \;=\; \sum_{i=k}^{W-1} 2^{i-k} b_i.$$

This interpretation of right shifts as division by powers of 2 is so useful that it is preserved even for negative numbers. And, just as filling to the left with `0`s for positive numbers doesn't change the nominal value of a binary number, so does filling to the left with `1`s for negative numbers in two's complement representation. Verify for yourself that if $x < 0$ then:

$$\mathsf{twoc}(x, W) \texttt{ >> } k \quad = \quad \mathsf{twoc}(\lfloor x/2^k \rfloor, W).$$

Bitwise Operators The primary use of shifting is to put a certain bit in a word in any other bit position in the word. Imagine we want to place the kth bit in the LSB position. We could shift the word right by k which would drop the k least significant bits and shift all the other bits k positions rightwards.

$$\texttt{0b}\,\underbrace{b_{W-1}\cdots b_k}_{W-k \text{ bits}}\underbrace{b_{k-1}\cdots b_0}_{k \text{ bits}} \texttt{ >> } k \quad = \quad \texttt{0b}\,\underbrace{b_{W-1}\cdots b_{W-1}}_{k \text{ bits}}\underbrace{b_{W-1}\cdots b_k}_{W-k \text{ bits}}$$

However, this still potentially leaves a lot of other unwanted bits in the word, including many copies of the sign bit. How would those be zeroed out as well? Shifting alone is not enough as the sign bit can never be zeroed out by shifting. Thus, we need some other operators that can modify individual bits.

You are already familiar with the Boolean operators `and`, `or`, and `not` from Python that produce new `bool` values from old values that can be interpreted as `bool`. We can draw up a *truth table* for these operators that show how they work on `bool` values.

b1	b2	b1 and b2	b1 or b2	not b1
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Since a bit can be either 0 or 1, we can think of them as a kind of degenerate `bool` value, with 0 standing for `False` and `1` for `True`. With this interpretation, we can build a *bitwise* version of the {`and`, `or`, `not`} operators that generates new bits from old bits in terms of their interpretation as `bool` values. These bitwise operators have slightly different names from their Boolean counterparts: bitwise *and* is written `&`, bitwise *or* is `|`, and bitwise *not* is `~`. (For now we are going to believe a white lie about `~`; we will get back to the full story towards the end of this subsection.) They have the following truth table.

b1	b2	b1 & b2	b1 | b2	~ b1
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

These operators are called bitwise operators because they actually work on entire words at a time, with the operation applied in parallel for corresponding bit positions. In other words, if `b1` and `b2` are W-bit words, then `b1 & b2` is a W-bit word where the kth bit is determined by `&`-int the kth bits of `b1` and `b2`.

To see how these operators are used, think back to our earlier goal of moving the kth bit of the word to the LSB position. We used right shift, but it did not zero out the other bits in the world. Well, we can zero them out by using the `&` operator and the value `0b1` with has `0`s for all by the LSB. Any bit `&`-ed with `0` will be `0`, while any bit `&`-ed with `1` will stay the same. Here, for instance, is the bit at index 3 of the 8-bit word 42, i.e., `0b00101010`:

```
>>> (0b00101010 >> 3) & 0b1
1
```

We can repeat this with all the bits to break the word up into a list of its bits.

```
>>> [(0b00101010 >> i) & 0b1 for i in reversed(range(8))]
[0, 0, 1, 0, 1, 0, 1, 0]
```

The relation between bitwise not (`~`) and mathematical negation (`-`) is interesting. Consider the meaning of `x + (~ x)`. If we perform the long sum, we will observe that all the bits end up being `1`. For instance, if $W = 8$ and $x = 42$, then we have:

$$\begin{aligned} \texttt{x} &= \texttt{0b00101010} \\ \texttt{~ x} &= \texttt{0b11010101} \\ \texttt{x} + \texttt{~ x} &= \texttt{0b11111111} \end{aligned}$$

But, a word with all bits set to `1` just represents -1 in two's complement representation. In other words, for finite words we have `x +~ x` $= -1$, or, rearranging,`- x =~ x + 1`. This is a much simpler definition of mathematical negation than the two's complement definition (Defn. 2.1). It is also simple to implement: to mathematically negate x, first flip all its bits and then add 1. Verify by yourself that this works regardless of whether x is positive or negative as long as we use the two's complement representation of negative numbers.

There is an additional bitwise operator that has no simple Boolean counterpart: *exclusive or* (*xor*). It is written with `^` and has the following truth table.

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

In other words, it returns `1` if and only if both its operands are different bit values. Another way to read this is that for any bit `b`, the bit `b ^ 1` is its opposite, so `1`-bits can be used to *toggle* the value of a bit position. In particular, if you xor a word `w` with a word with all `1`-bits, i.e., the word that represents `-1` in two's complement representation, then you get its bitwise negation:

```
w ^ (-1) == ~ w
```

Moreover, for any word `w`, it will be the case that `w ^ w == 0b0`.

PYTHON-SPECIFIC BITWISE NOT In Python, the type of integers **`int`** has an unbounded word size, meaning that it can store integers of arbitrary magnitude. This brief aside is very specific to Python, since nearly every other programming language, not to mention the processor hardware, only works with words of a finite word size.

When we apply bitwise not (`~`) to a positive Python integer, we obtain an integer whose bit representation has the reverse of the bits of the original number in the least significant bits, but all the most significant bits are `1`. For example:

```
  42 = 0b···0000000000101010
~ 42 = 0b···1111111111010101
```

This presents a problem: how could we represent an infinite number of `1`s to the left? *A priori* it would seem to require infinite space. Fortunately, we can use a neat trick about two's complement representations to get around this need for unbounded space. Recall that for finite word sizes we have already seen that `x+~ x =-1` is a provable identity. We can rearrange that to: `~ x =-(x + 1)`. The clever trick in Python is that, even for infinite W, we can consider this equation to hold, i.e., we will take it as a *definition* of bitwise not. This reduces the problem of doing bitwise not in finite space to doing mathematical negation (`-`) using finite space.

And here is how Python does just that: instead of using the MSB to store the sign bit, which is impossible if there is no most significant bit, Python just stores the sign bit separately. Conceptually, an integer is a 2-tuple of its *magnitude* and its *sign*. You can see this by using the **`bin`**`()` function on a negative argument.

```
>>> bin(-42)
'-0b101010'
```

Observe that the bit pattern stored is the same as that of the positive integer 42, i.e., `0b101010`. The fact that it is negated is depicted differently, with a `-` in front, instead of by modification of the bit pattern. For this reason, this representation is sometimes known as the *signed magnitude* representation. Mathematical negation of such numbers is extremely cheap (just flip the sign). Shifting left or right with such a representation is also simple: the sign is left unchanged, while the bit pattern is shifted left or right by the required amount.

In computer processors, signed magnitude representations of integers had a brief window of popularity in very early mainframe computers, such as the IBM 7090, but hasn't been seen since the mid 1960s. One reason to disfavor it is that it gives a representation for nonsensical integers such as `-0`. A more likely reason is that the other bitwise operations (`&`, `|`, etc.) are a lot more complicated and expensive to implement for negative integers (cf. Exercise 4.)

2.5 BIT HACKING

Given a bit `b`, we can force it to `0` (with `b & 0`), force it to `1` (with `b | 1`), and toggle it (with `b ^ 1`). How would we do this for arbitrary bit position(s) in a word? The idea is to create a precise pattern of bits—called a *bit mask*—and then combine them using bitwise operations to the input word.

The simplest case is for setting the kth bit to `1`. We just need to produce the following bit mask

$$\texttt{0b1}\underbrace{\texttt{00}\cdots\texttt{0}}_{k \text{ bits}}$$

This is our friend 2^k from before which we already know how to compute with left shifts. All that remains then is to do a bitwise or with this mask.

```
def set_bit(w, k):
    """set the kth bit of w to 1"""
    return w | (0b1 << k)
```

Toggling the kth bit is not much different, since xoring a bit with `0` leaves it unchanged.

```
def toggle_bit(w, k):
    """toggle the kth bit of w"""
    return w ^ (0b1 << k)
```

For setting the kth bit to `0`, which is usually called *clearing* the kth bit, the process is a bit trickier. We have to produce the following bit mask to `&` to the word:

$$\texttt{0b}\cdots\texttt{111}\,\texttt{0}\underbrace{\texttt{11}\cdots\texttt{1}}_{k \text{ bits}}$$

When trying to produce such bit masks where most bits are `1`, it helps to think of the complement of the mask. In this particular case, the complement is:

$$\texttt{0b1}\underbrace{\texttt{00}\cdots\texttt{0}}_{k \text{ bits}}$$

Obviously this is 2^k again. Here then is the full function.

```
def clear_bit(w, k):
    """set the kth bit of w to 0"""
    return w & (~ (0b1 << k))
```

99% of computer programmers will never need to know more about bit-level hacking than constructing and using such simple masks and using the set/clear/toggle operations. (See also Exercise 6.) Here are a small number of other famous tricks you can do with low-level programming with bits.

- Given two integers `x` and `y`, you can swap their values "in place" with the use of xor, avoiding any temporary variables:

```
x = x ^ y
y = x ^ y
x = x ^ y
```

  Try to figure out why this works using the basic fact that `x ^ x = 0` for any number `x`. Note that this technique is hardly ever useful in Python since you can just say:

```
(x, y) = (y, x)
```

- One of the most famous bit hacks is counting the number of `1`-bits in the binary representation of a non-negative number. The trick is to observe that `x & (x - 1)` has the same binary representation as `x`, except that the least significant `1`-bit in `x` is flipped to `0`. Try to figure out by yourself why this is so. We can then just count the number of times this can be done to `x` before it becomes `0`.

```
def count_ones(x):
    """Number of 1-bits in the binary representation of x"""
    k = 0
    while x is not 0:
        x &= x - 1
        k += 1
    return k
```

Hacking things at the bit level can be a lot of fun if you have a certain mindset, particularly if you want to work on very low level things such as microcontrollers, embedded software, or high performance graphics. Here are some additional resources:

- The *Bit Twiddling Hacks* page that lists a number of famous bit manipulation techniques.
- *Hacker's Delight* by Henry S. Warren. This is basically an encyclopedia of every bit hacking technique ever invented, regardless of whether it's useful or advisable. Don't invest serious time into this book unless you're sure that bit hacking is your passion.

2.6 OTHER BASIC TYPES: BOOLS, FLOATS, STRINGS

We have seen how to encode integers, even negative integers, using bit sequences. How would we represent other basic datatypes? In a standard computer processor there is usually at least one other primitive datatype for fixed floating point numbers, which we will cover soon. First, let us look at something even simpler: Booleans.

Booleans and Bitsets As we have already seen, a single bit is sufficient to encode a Boolean. However, computer processors and storage (memory, hard drives, etc.) are always given in units of bytes. We can of course always use the bit pattern `0b00000000` and `0b00000001` to store Booleans, but that is obviously extremely wasteful: only 12.5% of the space has any meaningful content.

In the worst case where we need to store individual Booleans, this overhead is unavoidable. However, it is often the case that we need to store several Boolean values at once. We can pack eight Booleans at a time into a single byte, as long as we are careful to remember which of the bits stand for which Boolean. This kind of packing is common in processor instructions where a number of *flags* need to be communicated to an instruction at once. It is also standard in very low level software libraries written in and for languages such as C, which provide useful language features to assemble and deconstruct such *packed* Booleans.

One interesting application of packed Booleans is to implement sets of small natural numbers, say numbers in $0, 1, \ldots, W-1$ where W is the word size. Imagine that a set is represented by an ordinary word. We would say that the number k is in the set if its kth bit is `1`; otherwise k is not in the set. Since we have already seen how to set and clear particular bits in Section 2.5, we can repurpose them to implement the set operations. These sets are thus known as *bitsets*.

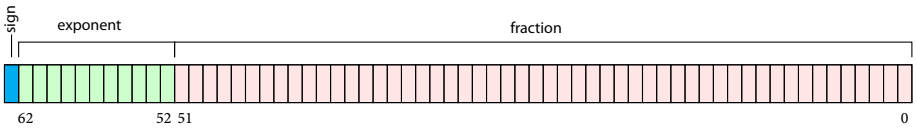
```
def empty_bitset():
    return 0

def bitset_mem(s, k):
    """Check if element k is in bitset s"""
    return ((s >> k) & 0b1) is 0b1

def bitset_add(s, k):
    """Add element k to bitset s"""
    return set_bit(s, k)

def bitset_del(s, k):
    """Remove element k from bitset s"""
    return clear_bit(s, k)
```

Floats Practically every computer today represents floating point numbers using the IEEE-754 standard for floating point arithmetic. The key feature of this representation of floats is it has *fixed precision*: it is only accurate over a small portion of the range of values that can be represented. (It is mathematically impossible anyhow to represent all real numbers with finitely many bits.) Like with signed integers, the MSB is reserved for the sign of the float, and the rest of bits of the word is broken up into the *exponent* and the *fraction*.

In this course we will never program with the binary representation of floats, but if you are really curious here is how 64-bit floats, called *double precision* floats or just *doubles*, look at the bit level. The 64 bits of the word are divided into:

- 1 bit for the sign (the MSB),
- 11 bits of *exponent*, and
- 52 bits of *fraction*

laid out like so:



The exponent is formed from bits 62, …, 52 of the 64 bit word, which represents an 11 bit integral value e between 0 and $2^{11} - 1 = 2047$. The fraction is formed from the bits 51, …, 0 of the 64 bit word; let these bits be $b_{51}, \ldots, b_0$. Consider the binary fractional number represented by $1.b_{51} \ldots b_0$ in base 2, i.e., the rational number $1 + F$ where F is given by:

$$F = \sum_{i=0}^{51} \frac{b_i}{2^{52-i}}$$

The sign s is given by the MSB which is either a 0 or a 1. Putting this together, the nominal value of this double precision floating point number is defined to be: $(-1)^s \times (1+F) \times 2^{e-1023}$ for nearly all bit patterns. A few special cases are interpreted as exceptions:

- If $e = 0$ and $F = 0$, then it represents either $+0$ or -0 depending on the sign s. This is sometimes called a *signed zero* and the two values are not considered equal.
- If $e = 0$ and $F \neq 0$, then the value is called a *subnormal*, which is a number so close to 0 that floating point operations lack precision and can introduce large errors. An operation that would produce a subnormal result is said to cause an *underflow*.
- If $e = 2047$ and $F = 0$, then the value represents $\pm\infty$ depending on the sign s.
- If $e = 2047$ and $F \neq 0$, then the value is consider invalid, and all such invalid values are represented by a special floating point literal `NaN` (standing for *not a number*).[2] Observe that a very large number of possible bit representations are invalid: indeed, there are $2^{52} - 1$ invalid numbers. Since `NaN` can stand for any one of these, we obtain paradoxical computational results such as `NaN != NaN`.

This encoding may seem unusually cumbersome at first glance, but it is designed to wring as much precision as possible while keeping the range of values as large as possible, while at the same time ensuring that arithmetic operations such as multiplication and logarithm can be computed efficiently. Covering such operations in more depth is beyond the scope of this course.

STRINGS A string is a sequence of characters, and each character is an integer that represents one of the indices in an extremely long array of possible characters. This array is known as *Unicode* and is intended to be as comprehensive as possible for languages that are currently used in the world—or have been used in the past—that might conceivably have a purpose in being recorded in digital form. This array is curated by the *Unicode Consortium* who publishes updates every few years or so; these updates add new characters to the end of the array, but never change any of the existing characters. Every index in the array is called a *code point*. As of version 11.0 of the Unicode standard, there are just under 140,000 characters; thus, each character can be encoded using $\lceil \log_2(140000) \rceil = 18$ bits. This means that all code points can easily be represented using 32-bit words, which is formalized in the UTF-32 encoding.

However, it turns out that most strings only use a small subset of this vast space of characters. Indeed, text that uses only Latin characters will use the smallest code points since these characters were added to the Unicode standard first and are therefore earliest in the space of code points. This means that most of the bits of the UTF-32 representation of the code points will be `0`. To counter this, modern encodings of strings do not use the same number of bits for every character. Small code points are represented with 8

[2] In Python you can construct a `NaN` value by using `float('NaN')`.

bits, while larger code points get 16, or 24 bits depending on their size.[3] This is formalized in the UTF-8 encoding of strings, which is the default encoding on the internet and in Python. To find the kth character of a UTF-8 encoded string, one has to start from the beginning and scan k characters forwards one-by-one; unlike with UTF-32, random access in the string is not $O(1)$. We will briefly return to the topic of compact string representations in the next chapter.

2.7 EXERCISES

1. Given W and a range of representable values $R = [-2^{W-1}, 2^{W-1})$, prove the following.
   - If $x, y \in R$ and $x + y \in R$, then $\mathsf{twoc}(x + y, W) = \mathsf{twoc}(x, W) + \mathsf{twoc}(y, W)$.
   - If $x, y \in R$ and $x - y \in R$, then $\mathsf{twoc}(x - y, W) = \mathsf{twoc}(x, W) + \mathsf{twoc}(-y, W)$.
   - If $x, y \in R$ and $x \times y \in R$, then $\mathsf{twoc}(x \times y, W) = \mathsf{twoc}(x, W) \times \mathsf{twoc}(y, W)$.

2. Define a python function `sm_to_twoc(x, W)` that takes `x` given in signed magnitude representation for a word size `W` and returns its two's complement representation for the same word size `W`. Map the special value `-0` to `0`.

3. Now define the reverse, i.e., a python function `twoc_to_sm(x, W)` that takes `x` given in two's complement representation for a word size `W` and returns its signed magnitude representation for the same word size `W`. Note that the value $2^{\mathtt{W}-1}$ in two's complement cannot be represented in signed magnitude, so return `None` for that case.

4. Use your solution to the previous two exercises to define bitwise `&` and `|` for signed magnitude representations. Can you do the same directly, without the loops embedded in the two functions?

5. Implement the function `is_pow2(x)` which returns `True` iff `x` represents an integer of the form 2^n for some n. You may assume that `x` is positive. See if you can do it without a loop by studying the `count_ones()` function from Section 2.5.

6. Generalize the functions `set_bit()`, `clear_bit()`, and `toggle_bit()` from Section 2.5 to work on entire bit masks. That is, write the following functions:
   - `set_mask(w, m)` that sets every bit position that is `1` in `m` to `1` in `w`.
   - `clear_mask(w, m)` that sets every bit position that is `1` in `m` to `0` in `w`.
   - `toggle_mask(w, m)` that toggles, in `w`, every bit position that is `1` in `m`.

   Do it without using any loops.

7. Recall that `%` is the modulus operator of Python, i.e., `w % k` returns the integer remainder when dividing `w` by `k`. Using what you know about bit masks, write the function `mod_pow2(w, k)` that computes the equivalent of `w % (2 ** k)` without using any exponentiation, division, or modulus operators.

[3] Technically, up to 32 bits can be needed to cover the full space of code points reserved by Unicode, but these 4-byte code points are not currently assigned to any characters.

3 RANDOMNESS AND SAMPLING

One of the most impressive uses of a computer is to perform experiments *in silico*, i.e., run experiments entirely on a computer. Unlike in the natural sciences, experiments on a computer have a lot of desirable features: it is easy to fully control the environment of an experiment, to repeat an experiment a large number of times, and to run many variants of an experiment with slightly different parameters. However, computerized experiments rely on the ability to make arbitrary choices, for otherwise a computer would always compute the same result. To make such arbitrary choices, we need *randomness*.

3.1 DISTRIBUTIONS AND SAMPLING

There is no such thing as a random number. Any specific number such as 42 or $-157.\overline{3}$ or e^{π} is no more "random" than any other specific number. *Randomness* is a property of the process of *selection* (aka *sampling* or *generation*) of a number from a range of possible numbers. The selection process can pick all numbers in the range with equal probability, or it can prefer some numbers to other numbers. In general there is a *distribution* of probabilities for the numbers produced by a selection process. When we talk about a "random number" in computer science, we implicitly mean a *number selected from a given range with a given probability distribution*, even if the range and the distribution are not explicitly stated.

The Python standard library module `random` contains many of the building blocks of random number generation. It has to be imported as usual with:

```
import random
```

UNIFORM DISTRIBUTION The most basic operations of the `random` module involve the *uniform distribution* that assigns the same probability of selection to all elements of its range.

- `random.randint(lo, hi)`
  Uniformly selects an `int` between `lo` and `hi`, with both ends included in the range.
- `random.random()`
  Uniformly selects a `float` in the range $[0.0, 1.0)$.
- `random.uniform(lo, hi)`
  A generalization of `random.random()`: generates a uniformly selected `float` in $[\texttt{lo}, \texttt{hi})$ or $[\texttt{lo}, \texttt{hi}]$, depending on rounding
- `random.choice(seq)`
  If `seq` represents a finite and non-empty collection of elements, then this function selects an element of this sequence uniformly. Note that `seq` can be used for anything that can be iterated over, so it is fine to use the `range()` function. Thus, the `random.randint()` function may have been implemented in the `random` module as:

  ```
  def randint(lo, hi):
      return choice(range(lo, hi + 1))
  ```

Biased and ranked choice The uniform distribution is the building block for a wide variety of derived random number generators that make biased choices. Perhaps the most common such scenario is when one needs to pick between two choices, A and B, with probabilities p and $1 - p$ respectively. This is easy to implement with `random.random()`:

```
def pick(A, B, p):
    """return A with probability p, and B with probability 1 - p"""
    if random.random() < p:
        return A
    return B
```

This works because the uniform distribution over a dense range such the real interval $[0, 1)$ has the property that the probability of a selected number falling in a subinterval such as the prefix $[0, p)$ is exactly the ratio of the sizes of the subinterval to the whole interval, i.e., p in this case. In fact, this is how uniform distibutions are defined on dense intervals in the first place, because the probability of selecting any individual number in the range is 0.

Ranked choice A generalization of biased choice between two items is *ranked choice* that picks from a finite ordered sequence of items with varying probabilities by making a sequence of biased choices until the first one succeeds. Given n items $[A_0, A_1, \ldots, A_{n-1}]$ and probabilities $[p_0, p_1, \ldots, p_{n-1}]$ (with $0 \leq p_i \leq 1$ for each i), ranked choice makes the following sequence of biased choices.

1. Choose A_0 with probability p_0.
2. If A_0 is not chosen, choose A_1 with probability p_1.

⋮

k. If $A_0, A_1, \ldots, A_{k-2}$ are all not chosen, choose A_{k-1} with probability p_{k-1}.

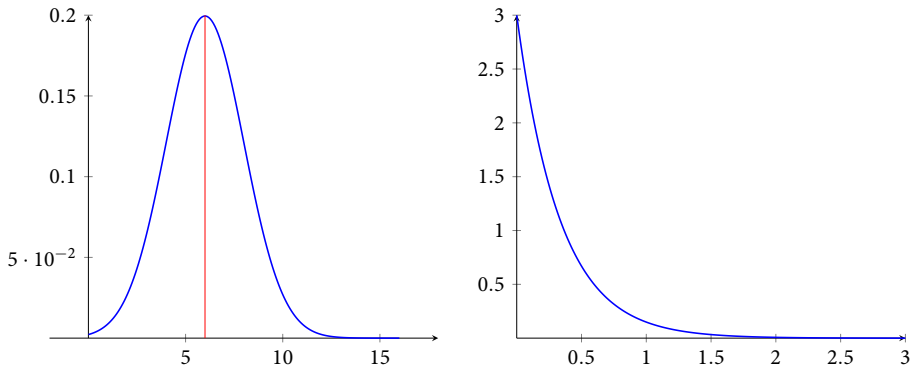
⋮

n. If $A_0, A_1, \ldots, A_{n-2}$ are all not chosen, choose A_{n-1} with probability p_{n-1}. It is a common convention to set $p_{n-1} = 1$ to guarantee that the overall choice always definitely chooses something, i.e., the last element if all the other ones are not chosen. With all the $p_i < 1$, the ranked choice function is partial, i.e., it is possible that none of the possibilities are chosen.

Whenever a choice is made, the operation ends. Thus, A_k is chosen with overall probability $p_k \prod_{i=0}^{k-1}(1-p_i)$, which accounts for choosing A_k and not choosing anything earlier. If all the p_i are the same value, say p, then A_k is chosen with probability $p(1-p)^k$.

Continuous non-uniform distributions There are a number of other distributions of interest if you want to perform statistical experiments with a continuous range. Two famous ones are:

- `random.normalvariate(mean, stdev)`
  Selects a **`float`** from a normal distribution with mean `mean` and standard deviation `stdev`. The normal distribution is very common when looking at collections of outcomes at a time instead of individual outcomes. By the *central limit theorem* of statistics, the sum of independent variables has a normal distribution, no matter what the distributions are of the variables themselves. Thus, when you run an experiment a number of times and compute the average values of a measurement, you would expect that average value to itself have a normal distribution.



(a) Normal (b) Exponential

Figure 3.1: Probability density function for some exponential distributions.

- `random.expovariate(ex)`
  Selects a **`float`** from an exponential distribution with parameter `ex`. The exponential distribution arises commonly in *memoryless* systems where the future is independent, in some sense, of the past. Common examples are the distribution of mean times between discrete events such as nuclear decay, comet strikes, queue arrivals, etc.

Figure 3.1 shows a plot of the *probability density function*[1] for these two distributions. Generally speaking such distributions are built out of the uniform distribution by various statistical processes that are beyond the scope of this course.

3.2 GENERATING RANDOM NUMBERS

We have seen *what* to call in Python to generate random numbers from uniform distributions, but *how* does Python *generate* these random numbers in the first place? All processor instructions that perform mathematical operations (addition, multiplication, etc.) or program control (conditionals, loops, etc.) are deterministic – they do the same thing every time they run. Because we want results that vary statistically, we are forced to depart from this abstract and deterministic world and start interacting with the real physical world, which is inherently statistical. One useful approach would be to *measure* real things: "wall clock" time, bytes communicated, user input, mains voltage, and so on. Such measurements tend to have a lot of noise, particularly in their least significant bits, which we can repurpose as a source of random bits. We can go even further with special hardware to extract random bits from physical processes: for instance, a Geiger counter to measure nuclear decay events, or a radio receiver to sample atmospheric noise.

Unfortunately, there are many inter-related problems with such *hardware random number generators* (HRNG). First, physical sources of randomness are rate-limited. Real computational uses of randomness require millions of uniformly sampled bits per second, whereas HRNGs can only deliver on the order of thousands of bits per second. Second, and more important, the bits that such a source generates may not be uniformly distributed. Worse, even if it starts out generating roughly even amounts of `1`s and `0`s, over time the bias may change as physical conditions change. Thus, HRNGs have to be continuously monitored during their lifetime, and countermeasures must be put in place to ensure that the bias does not affect the desired distribution of the generated bits.

[1] For a continuous random variable X with range $\mathbb{R}$, its probability density function $f_X : \mathbb{R} \to [0, 1)$ is defined such that for every $a, b \in \mathbb{R}$ with $a \leq b$, it is the case that $P[a \leq X < b] = \int_a^b f_X(x)\, dx$.

Implementation	m	a (multiplier)	c (increment)
ANSI C (2018)	2^{31}	1103515245	12345
Microsoft Visual C++	2^{32}	214013	2531011
Java	2^{48}	`0x5deece66d`	11
Musl	2^{64}	6364136223846793005	1

Table 3.1: Some common values for LCG parameters

PSEUDORANDOM NUMBERS To address these shortcomings of HRNGs, it is much more standard to use specific deterministic algorithms to generate *sequences* of bits that *appear to be* random—i.e., that satisfy statistical tests of randomness—but that can be executed at the full speed of the processor and without any need of external monitoring or bias reduction. Such deterministic algorithms are known as *pseudorandom number generators* (PRNG). Numbers extracted from chunking up the pseudorandom sequence into bytes are called *pseudorandom numbers*.

A PRNG has an internal state that gets updated with every request for a random bit. This state is initialized or *seeded* with a particular state, and the entire pseudorandom sequence can be recreated by just reusing this seed state. This is what makes a pseudorandom sequence not truly random, because knowing the seed guarantees that you can perfectly predict the entire sequence. To have a high chance of producing a fresh pseudorandom sequence that has never been seen before, this seed state must itself be unpredictable; for instance, it can be selected using a HRNG, or it can be set to the current value of some rapidly changing measurement such as the number of nanoseconds since January 1, 1970.

If a PRNG has a state of n bits, then it can generate no more than 2^n separate bit sequences; hence, every pseudorandom sequence will eventually loop. The number of bits before the sequence loops is called its *period*. The longer the period, the better the PRNG is, but this is not entirely without caveats. For instance, a PRNG that just increments the state by 1 each time it is called will iterate through the entire range without repetition and therefore have a very long period; however, it is perfectly predictable and useless. PRNGs that have a lot of state tend to have a longer period, but they also tend to be more complex algorithmically and therefore produce random numbers at a slower rate. Thus, there is generally a trade-off between the quality and the complexity of a PRNG.

EXAMPLE: LINEAR CONGRUENTIAL GENERATOR One of the oldest pseudorandom generation algorithms is the *linear congruential generator* (LCG) which generates m-bit numbers at a time and uses a single m-bit number as its state. Generally, $m = 2^{32}$ or $m = 2^{64}$ or very close, so that entire machine words can be generated at a time. The $(n+1)$th number generated by an LCG is a linear combination of its previous state and two m-bit parameters: the *multiplier* $a \neq 0$ and the *increment* c. More formally, an m-bit LCG sequence $X_0, X_1, \ldots$ starts with the initial (seed) value X_0, and has the following relationship between every adjacent pair of states:

$$X_{n+1} \equiv a X_n + c \pmod{m}$$

Clearly the choice of a and c is very important: if $a = 1$ and $c = 1$, then we just get a very non-random sequence of numbers increasing by 1 each time. Some standard choices for the parameters a, m, and c are shown in Table 3.1.

ADAPTING UNIFORM SAMPLES Suppose that we have a random number generator that uniformly selects an integer in the range $[0, \mathsf{n})$. How would we use it to uniformly generate an integer from a different

range, say $[\texttt{j}, \texttt{k})$? Easy! Just generate $\texttt{u} \in [0, n)$ and return

```
j + u * (k - j) // n
```

This simply scales and shifts the range $[0, \texttt{n})$ to $[\texttt{j}, \texttt{k})$. However, since integer division rounds down and therefore loses information, this kind of scaling only makes sense if $\texttt{n} \gg (\texttt{k} - \texttt{j})$.

A more interesting question is how to generate random `float`s. We could repeat the same trick as before by scaling the range but using floating point division (`/`) instead of integer division (`//`). As long as the source of random integers has a sufficiently long period, this kind of scaling tends to be reasonably strong; indeed, it is how the Python standard library implements `random.random()`. It is also possible—but very rare—to use a generator of bit sequences to directly generate random numbers in the IEEE-754 format (see Section 2.6).

Removing bias by rejection sampling Any process that makes consistently biased choices can be used to produce unbiased (i.e., uniform) choices by running the process repeatedly and looking at the probabilities for sequences of outcomes.[2] Consider the case of a biased coin that lands heads (`H`) with probability p and tails (`T`) with probability $1 - p$. Even if $p \neq 0.5$,[3] we could still use this coin to produce an unbiased coin toss, i.e., generate `H` or `T` with probability 0.5 each. How? Think about ordered pairs of tosses of this coin: there are four outcomes, `HH`, `HT`, `TH`, and `TT`, with the following probabilities:

$$\texttt{HH} : p^2 \qquad \texttt{TT} : (1-p)^2 \qquad \texttt{HT} : p(1-p) \qquad \texttt{TH} : (1-p)p$$

Obviously, no matter what p is, the probability of the sequence `HT` is the same as that of `TH`. We can extract an *unbiasing* algorithm from this observation.

1. Toss the biased coin twice.
2. If the outcomes are the same, go back to step 1; otherwise, return the *first* outcome.

There is nothing special about the first outcome in step 2. We just need to bijectively map `HT` and `TH` to `H` and `T`, so we arbitrarily pick the first; we could have picked the second without any problem. Here's how it looks in Python, assuming there is a function `biased_coin()` to generate biased coin tosses.

```
1 def unbiased_coin():
2     while True:
3         outcome1 = biased_coin()
4         outcome2 = biased_coin()
5         if outcome1 is outcome2: continue
6         return outcome1
```

The general technique illustrated by this example on unbiasing coins is known as *rejection sampling*. The idea is to repeatedly generate sequences of outcomes, looking for particular sequences whose probability is predictably uniform. Any sequence whose probability is not uniform can be rejected for a fresh run of the loop. Most often the sequences one looks for in rejection sampling is a *permutation* of the range (as was the case for coins above), because the probability of obtaining any permutation of the range is the same. This can be very expensive, though. The probability of obtaining a permutation of the range can be very small, and nearly every sequence could get rejected. This is why rejection sampling is most

[2] This assumes that repeated runs of the biased process are independent, which may or may not be a safe assumption, but we will ignore this issue for now.

[3] And as long as $p \neq 0$.

commonly used with the range $\{0, 1\}$ which generates a uniformly sampled stream of bits, which is then assembled into bytes, words, and other high-level data. Uniformly generated numbers from this process can then be adapted as described above.

3.3 SIZE ESTIMATION: MONTE-CARLO SAMPLING

As a first example of computerized statistical experiment, let us look at the problem of *size estimation*. Suppose you have a description of a region of space together with what counts as its interior, but you don't know what its size is. For instance, points (x, y) in the interior of the unit circle centered at the origin are easily described with the inequation $x^2 + y^2 \leq 1$. What is the area of this circle? Of course we know from middle school math that this will be exactly $\pi \cdot 1^2 = \pi$, but let's estimate this area with random numbers.

The technique we will use is known as *Monte-Carlo sampling*.[4] We know how to uniformly sample numbers in $[-1, 1)$ using `random.uniform(-1, 1)`. How do we uniformly select a point in a square of side 2 centered on the origin? Well, for any point (x, y) in this square, we know that the coordinates x and y are independent: neither is determined or constrained by the other. That is to say, if $x = 0.5$, say, then y can be any number in $[-1, 1)$ without going out of the square. So, it suffices to independently select both coordinates in the range $[-1, 1)$ by calling `random.uniform()` twice.

Now that we know how to uniformly select points in this square, let us select a large number of such points, say $N = 100,000$ points. Some of these points will fall inside the unit circle, while others will be outside the circle. Since the selection of points is uniform, by definition the probability of selecting a point inside the circle is the same as the ratio of the area of the circle to that of the enclosing square, which is $2^2 = 4$. Thus, if k of the N points are inside the circle, then we should expect that: $k/N \approx C/4$, where C is the area of the circle. Since we know analytically that $C = \pi$, we arrive at the numerical approximation $\pi \approx 4k/N$.

Here's how it looks in Python.

```
def estimate_pi(N=100_000):
    """Give a statistical estimation of pi"""
    k = 0 # Number of "successes"
    for _ in range(N):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if x ** 2 + y ** 2 <= 1:
            k += 1
    return 4 * k / N
```

```
>>> estimate_pi()
3.13604
>>> estimate_pi(1_000_000)
3.14106
>>> estimate_pi(10_000_000)
3.14156
```

As you can see, the quality of the estimate increases with the number of experiments run. As an estimation of π, this is far too slow and error-prone, but this is only a pedagogical example. Size estimation with Monte-Carlo sampling is generally best used in situations where an analytic answer is unavailable or difficult to construct, particularly in higher dimensions.

[4] Monte Carlo is famous for its casino and gambling. In computer science and statistics, any technique that relies on essentially generating and testing random numbers is given the descriptor "Monte Carlo", whether or not it directly deals with gambling.

To summarize, the overall technique for estimating the size of a space is:

1. Find an enclosing space whose size is known and is easy to sample uniformly,
2. Sample many points and count the ones that fall within the target area (the "successes"), and
3. Return the ratio of the number of successes

Inverse transform sampling When trying to pick the enclosing space for size estimation, it is important to keep in mind that it needs to be sampled uniformly. For instance, suppose we pick the enclosing area to be a circle of radius R. How do we uniformly sample a point in this circle? One way is to just use rejection sampling with an enclosing square (like we did above), but this is wasteful because it discards $\pi/4 \approx 25\%$ of the randomly generated points.

A different approach that often works with algebraic probability distribution functions is *inverse transform sampling*. This is largely out of scope of this course but we can see a small glimpse of it here. Suppose we want to generate a number X in the interval $[0, R)$ with a cumulative distribution function F_X, i.e.,

$$P[X \leq x] = F_X(x) \text{ for all } x \in [0, R).$$

Uniformly generate a number $U \in [0, 1)$ and consider solving for X such that $F_X(X) = U$. The solution is $X = F_X^{-1}(U)$, where F_X^{-1} is the inverse of F_X. This X has the desired cumulative distribution F_X:

$$\begin{aligned} P[X \leq x] &= P[F_X^{-1}(U) \leq x] && \text{by definition of } X \\ &= P[U \leq F_X(x)] && \text{apply } F_X \text{ to both sides} \\ &= F_X(x) && \text{because } U \text{ was generated uniformly.} \end{aligned}$$

This property of the inverse of the cumulative distribution function is sometimes called the *golden rule*. Note that not all functions are easy to invert algebraically.

To give a concrete instance, imagine we are trying to generate a point in the circle with polar coordinates of the form (r, θ). The angle θ can be uniformly sampled in $[0, 2\pi)$ (measured in radians), but the radius r cannot be uniformly sampled in $[0, R)$ because the sampled range (area) grows quadratically with the radius. Indeed, the probability that the point is within a distance k from the center is the ratio of the areas of the disc of radius k to that of the full circle of radius R, which is $(\pi k^2)/(\pi R^2) = (k/R)^2$. So:

$$P[r \leq k] = \left(\frac{k}{R}\right)^2 \text{ for all } k \in [0, R)$$

This is our cumulative distribution function $k \mapsto (k/R)^2$; its inverse is, obviously, $u \mapsto R\sqrt{u}$. Thus, the algorithm to generate a uniformly sampled polar coordinate in a circle of radius R is as follows; uniformly sample $u \in [0, 1)$ and $\theta \in [0, 2\pi)$ and return the coordinate $(R\sqrt{u}, \theta)$.

Let's use this insight to estimate the area of the *cardioid* given by the polar inequation $r \leq 2(1 - \cos\theta)$, which is the red region depicted in Figure 3.2. A circle of radius 4 centered at the origin is shown in green. Sampling points uniformly in the circle amounts to generating a uniform sample in $u \in [0, 1)$ and then returning $4\sqrt{u}$ for the radius. The full area estimation of the cardioid is shown in Figure 3.3. Since the circle itself has area 16π, the estimate returns the fraction of it that corresponds to successfully sampled points in the interior of the cardioid. Note that the expected area of the cardioid is:[5]

$$\iint_{\substack{0 \leq \theta \leq 2\pi \\ 0 \leq r \leq 2(1-\cos\theta)}} r \, dr \, d\theta = 6\pi.$$

[5] Don't worry if you don't have the mathematical background to perform that double integral yourself. This course doesn't require multivariable calculus, so you won't be expected to perform such analytical calculations.

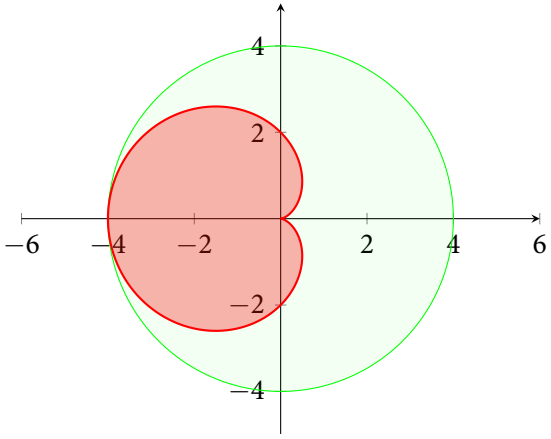


Figure 3.2: A cardioid and its concentric circle

```
def estimate_cardioid(N=100_000):
    k = 0
    for _ in range(N):
        r = 4 * math.sqrt(random.random())
        th = random.uniform(0, 2 * math.pi)
        if r <= 2 * (1 - math.cos(th)):
            k += 1
    return 16 * math.pi * k / N
```

```
>>> estimate_cardioid()
18.889768307504706
>>> 6 * math.pi          # This is the expected result
18.84955592153876
```

Figure 3.3: Estimating the area of a cardioid

Hence, we expect 6/16 of the sampled points to fall inside the cardioid.

3.4 TESTING STATISTICAL INTUITIONS

Statistical mathematics can often be complex and produce counterintuitive results. With a computer you can sometimes *simulate* the statistical problem directly and run a large number of experiments to estimate probabilities, just like we estimated sizes with Monte-Carlo sampling earlier. Testing intuitions by programming is a powerful tool in the toolbox of every statistician.

Let us see it in action with a classic puzzle that deals with the question of the emergent sex ratio of a population that follows a certain breeding protocol. Suppose we have a population of immortal creatures where each creature has exactly one of two sexes (`M` or `F`). Assume that only heterosexual couples can produce children, and that each child can turn out to be either `M` or `F` with equal probability independent of all other birth events. In this population, each couple follows the following protocol: they continue to produce offspring until they have an `M` child, at which point they stop reproducing. For example, one couple could stop on their first child if it is an `M`, while another couple could produce the child sequence `FFFFFFM`. Given such a population of k couples, what would be the eventual expected sex ratio of children?[6]

[6]This is sometimes called the *royal families puzzle* because of the apparent similarity of this breeding protocol to that of

Here are three intuitive answers:

- There will be more `F`s than `M`s, because enough couples will have several `F` children, while all couples will have only one `M` child.
- There will be more `M`s than `F`s, because each couple will eventually definitely have an `M` child while there will be couples without an `F` child.
- The expected sex ratio will be 50:50 because each birth event individually produces an `M` or an `F` child with equal probability.

Which answer is correct? Before reading further, try to figure it out by yourself analytically.

We can easily program this scenario and compute the averages to see if our intuitions about the problem are correct. Let us use a counter `num_f` to count the number of `F` children produced. Then, for each couple we have them repeatedly produce offspring until they produce an `M` child, incrementing `num_f` every time they get an `F` child. Since the number of `M` children will be the same as the number of couples k at the end of the run, the sex ratio will be easily computed. We can then rerun this whole simulation `N` times and compute the average sex ratio over `N` runs.

Figure 3.4 has the code for the problem with some example values of the number of couples k. If we plot it out (Figure 3.5), we will see that the ratio will get asymptotically closer to 50% as $k \to \infty$, but will always stay less than 50%. If you predicted that the expected eventual sex ratio would be 50:50, then you probably committed a classic error about combining expected values. Expected values are additive, not multiplicative: $E(X/(X+Y)) \neq E(X)/E(X+Y)$. The right hand side of that disequation would be 0.5 if X represents `F` and Y represents `M`, but we wanted the figure on the left.

3.5 EXERCISES

1. Many board games make use of one or several dice to determine specific outcomes. The result of a specific outcome is often specified using the format *ndk* which stands for rolling (or casting) n fair dice each of which has k sides numbered $1, \ldots, k$, and then adding up the results. For instance `3d6` means to cast three 6-sided dice and return their sum, i.e., the effect of running:

   ```
   random.randint(1, 6) + random.randint(1, 6) + random.randint(1, 6)
   ```

   Write the function `roll_dice(n, k)` that returns an `ndk` roll.

2. Suppose you are given a function `biased_die3()` that returns `0`, `1`, or `2` with unknown—but fixed—probabilities.[7] How would you use this function to generate bits uniformly? Write the function `unbiased_die2()` that returns `0` or `1` with equal probability; the only function it is allowed to call is `biased_die3()`.

3. Suppose you are given a biased 6-sided die function `biased_die6()` that generates numbers in `[0, 1, 2, 3, 4, 5]`, with number having a fixed but not necessarily uniform probability. Use this function to write an unbiased 6-sided die function `unbiased_die6()` that generates numbers in the same range uniformly.

4. One standard way to generate a pseudorandom sequence of bits is with a *linear-feedback shift register* (LFSR) that uses a single W-bit word as its state. Certain bit positions in the current state are *tapped*,

historical royal families.

[7] A three-sided die looks like a triangular pencil. To cast the die, you would roll it and see which face ends up on the bottom.

```
def royal_family_1(k):
    """Simulate one set of k royal couples. Returns the ratio of
    F-children to all children."""
    num_f = 0
    for _ in range(k):
        while random.randint(0, 1) is not 0:
            num_f += 1
    return num_f / (k + num_f)

def royal_family_n(k, N=100_000):
    """Simulate N sets of k royal couples and compute the average
    ratio of F-children"""
    f_ratio = 0
    for _ in range(N):
        f_ratio += royal_family_1(k)
    return f_ratio / N
```

```
>>> royal_family_n(1)
0.3061527875976661
>>> royal_family_n(2)
0.3859145573710632
>>> royal_family_n(3)
0.4221397618610884
...
>>> royal_family_n(100)
0.49750764849191487
```

Figure 3.4: Implementing the royal families puzzle

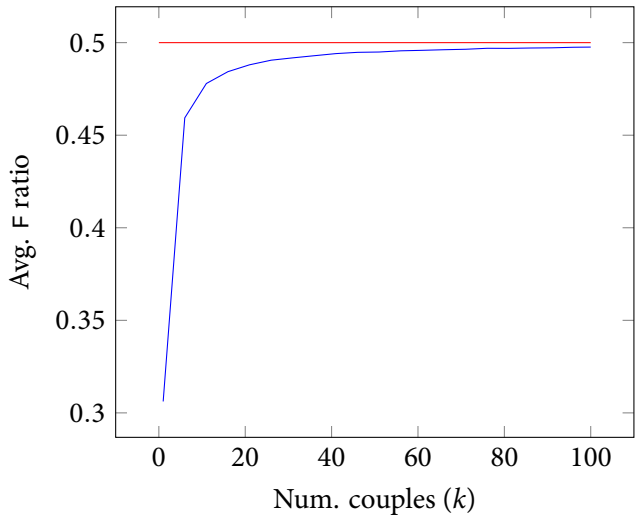
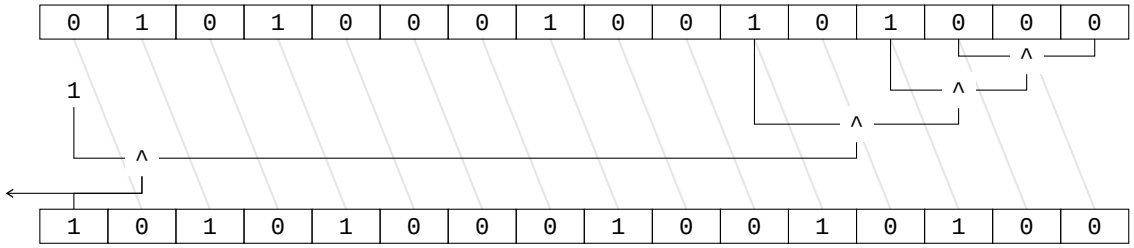


Figure 3.5: Plot of the expected F-ratio for various values of k.

which is to say that their bits are extracted, and then combined together get the next random bit. Usually the combination is exclusive or (`^`) together with an extra `1`-bit. The state is then shifted right by one and the most significant bit of the state is set to the random bit computed before. For instance, if $W = 16$ and we tap positions 0, 2, 3, and 5, then we observe the following sequence of changes for the given start state.



Write the following functions:

- `tap(W, state, b)`: extracts the bit at position `b` in `state`, which is assumed to be `W`-bits wide. Remember that bit positions are numbered starting from 0 and going right to left.
- `polytap(W, state, bs)`: taps all the bit positions in `bs` in `state` (again assumed to be `W`-bits wide), and xors them together with `1`.
- `lfsr(W, state, bs)`: returns a pair `(nstate, b)` where `b` is the bit produced by `polytap(W, state, bs)`, and `nstate` is `state` shifted right by 1 and with `b` at bit position `W`.

5. (*Monty Hall problem*) One of the classic puzzles of probability theory is based on a famous American game show hosted by Monty Hall. The setup is as follows: you are given a choice of three doors. Behind one door is a car, while behind the other two are goats. You guess which door hides a car. In response, Monty picks one of the two other doors that contains a goat and reveals it. He then asks you if you want to change your chosen door. Should you change your choice to increase your chances of getting the car?

   If you've never seen this problem before, then first try to figure the answer out analytically. Then, write a program to simulate it. To be precise, make the following assumptions about the setup, Monty, and yourself (the contestant):

   - The initial setup is uniformly random, as is the initial choice by the contestant.
   - After the first choice, Monty uniformly picks one among all the doors that hide goats. That is to say, if the initial guess was the car, then Monty picks one of the other two doors uniformly, but if the initial choice was a goat then Monty always reveals *the other* goat.

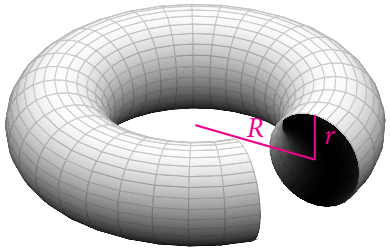
6. As you're walking down the street you run into M. Dupont who introduces you to his daughter walking with him. You know that M. Dupont has two children. What are the chances that his other child is a boy? Assume all random aspects of this setup are uniform. Make a conjecture analytically, then simulate the setup and verify that your conjecture is valid.

7. (*Birthday paradox*) Number the days of a year from 0 to 364. (To simplify matters, we'll assume all years have 365 days.) Assuming that people are born uniformly throughout the year, what is the size of the smallest group of people for which the probability that at least two of them were born on the same numbered day is at least 0.5? Program a simulation that tries various threshold probabilities in addition to 0.5. Can you spot how the group sizes and the threshold probabilities are related?

8. Here is an alternative algorithm for sampling points inside a unit circle.
   (a) Pick two angles θ_1 and $\theta_2 \in [0, 2\pi)$ uniformly, retrying in a loop until $\theta_1 \neq \theta_2$.
   (b) Return the midpoint between the two points indicated by the two corresponding radii, i.e., between the points with Cartesian coordinates $(\cos\theta_1, \sin\theta_1)$ and $(\cos\theta_2, \sin\theta_2)$.

   Is the distribution of points generated by this algorithm uniform?

9. (*Volume of torus*) A torus is the well known tube of inner radius r arranged in a ring of radius R, as shown in the following figure.



   Imagine that the torus is laid out in the x, y plane with the center of its outer circle at the origin. It can be inscribed in a cuboid with opposite corners $(R + r, R + r, r)$ and $(-(R + r), -(R + r), -r)$.

   - First, find a way to sample a point (x, y, z) uniformly within this cuboid.
   - Given such a point (x, y, z), it will fall inside the torus if:

$$\left(\sqrt{x^2 + y^2} - R\right)^2 + z^2 \leq r^2.$$

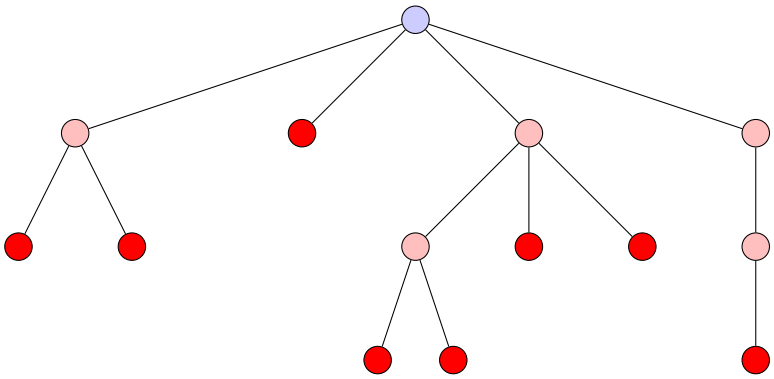
   - Use the above fact to generate a large number N of uniformly sampled points and count the number that fall within the torus. Use this to estimate the volume of the torus.
   - Analytically, the volume of the torus is obviously $(\pi r^2)(2\pi R) = 2\pi^2 r^2 R$.

   Concretely, write this as the function `toroidal_volume(R, r, N=100_000)` that returns an estimate of the volume of the torus of inner radius `r` and outer radius `R`. How many samples do you seem to need to get to within $\pm 5\%$ of the analytically determined volume?

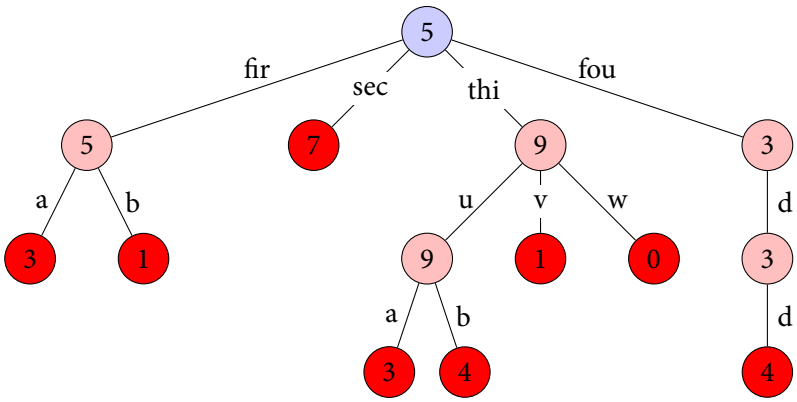
4 TRAVERSING TREES

Given a data structure, a natural operation to perform on it is to *traverse* it, which roughly means to move around in the contents of the structure by making use of the relationships inherent in the structure. We have already seen this in the context of *linear* structures such as lists where we can iterate over the elements of the list front-to-back or back-to-front, making use of the linear ordering. In this chapter we will be concerned with *hierarchical* instead of linear structures.

In particular, we will consider *trees*, a structure that you saw briefly at the end of CSE 101. Trees consist of *nodes* of two kinds: *leaf nodes* (colored red in the figure below) that contain no descendants of their own, and *internal nodes* (colored pink) which can have one or more trees as descendants. A single tree can be unambiguously referred to by its *root node* (colored blue); this node refers to all its immediate descendant trees—called *subtrees*—by their root nodes, which in turn refer to their subtrees, and so on.



To keep things relatively concrete, we will work with a tree data structure where at each kind of node, both internal and leaf, there will be a `data` field storing some data. In addition, to keep the subtrees of an internal node distinct, we will store them in a dictionary where the keys are *labels* of the branches of the tree, and the values are the subtrees referred to by the corresponding edges. For instance, imagine the above tree where each node contains an integer element, and the edges are labeled by strings. It may look as follows. (Note that edge labels need not be unique over the whole tree.)



PYTHON REPRESENTATION We will encode nodes using the `Node` class, whose initializer will take the data and the dictionary mapping edge labels to subtrees as the `kids` argument.

```
1  class Node:
2      def __init__(self, data, kids):
3          self.data = data
4          self.kids = kids
```

The above example tree is then constructed as follows:

```
>>> example_tree = \
... Node(5, dict(
...    fir=Node(5, dict(
...        a=Node(3, dict()),
...        b=Node(1, dict()))),
...    sec=Node(7, dict()),
...    thi=Node(9, dict(
...        u=Node(9, dict(
...            a=Node(3, dict()),
...            b=Node(4, dict()))),
...        v=Node(1, dict()),
...        w=Node(0, dict()))),
...   fou=Node(3, dict(
...       d=Node(3, dict(
...           d=Node(4, dict()))))))
...
>>> example_tree
<__main__.Node object at 0x7fca2a15b550>
```

MEASURING THE TREE A path from the root of the tree to a leaf can be constructed by simply collecting the edge labels into a list. For instance, the path from the root to the leaf containing the data 0 is the list `['thi', 'w']`. The *length* of such a path is just the length of this list representing the path.

Definition 4.1 (Depth). *The* depth *of a tree is the length of a longest path from the root to one of its leaves.*

This is fairly easy to compute by recursion.

```
1  def depth(root):
2      """Return the depth of the tree rooted at the given node"""
3      if len(root.kids) == 0: return 0
4      return max(map(lambda subtree: 1 + depth(subtree),
5                     root.kids.values()))
```

Line 3 considers the case that the tree is just a single node, in which case its depth is 0. Otherwise, in line 4–5 we map each of the subtrees to its depth, incremented by 1, and then compute the maximum of this list of numbers. The depth of the `example_tree` above is 3.

Another important measure of a tree is its *breadth*, which measures how "wide" the tree is.

Definition 4.2 (Breadth). *The* breadth *of a tree is the maximum number of nodes that are at the same path length from the root of the tree.*

This can be computed in a variety of ways. For instance, one can write a function that returns all the nodes that are at a given distance from the root of the tree; the breadth is then the maximum length when considering distances from 0 to the depth of the tree. (See Exercise 1.) The breadth of the `example_tree` is 6, which corresponds to the nodes that are at distance 2 from the root.

4.1 DEPTH-FIRST TRAVERSAL

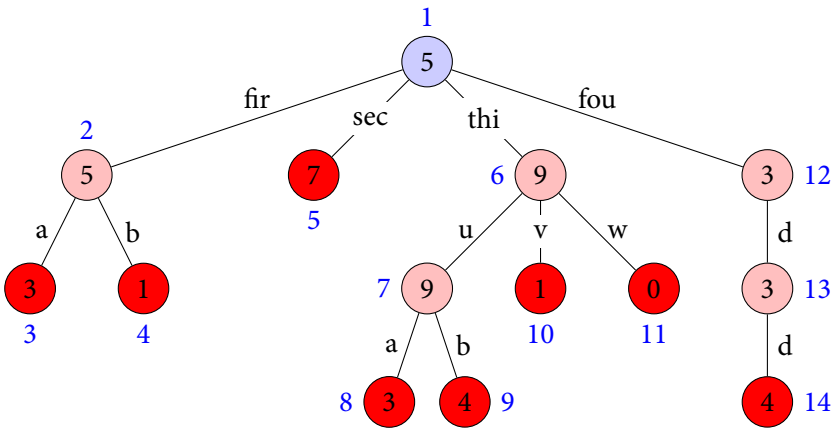
A *traversal* of the tree can be seen abstractly as an order of *visiting* the subtrees of a tree. *Depth-first traversal* (DF) has the property that all the nodes in a subtree are visited before any sibling subtrees are considered. Usually depth-first traversal is used in *depth-first search* (DFS), which is a search algorithm that tries to find an element in the tree by means of exhaustively searching a given subtree before moving on to a sibling subtree, and doing this hereditarily for every subtree.

The basic pattern in a depth-first traversal is best illustrated with recursion. Here, for instance, is a function that simply prints a message saying that a given node has been visited.

```
1 def df_visit(root):
2     """Visit the tree that is rooted at the given root"""
3     print('Visiting {}, containing data {}'
4           .format(root, root.data))
5     for kid in root.kids.values():
6         df_visit(kid)
```

```
>>> df_visit(example_tree)
Visiting <__main__.Node object at 0x7fca2a040da0>, containing data 5
Visiting <__main__.Node object at 0x7fca2a1595c0>, containing data 5
Visiting <__main__.Node object at 0x7fca2a02d4e0>, containing data 3
Visiting <__main__.Node object at 0x7fca2a15bd30>, containing data 1
Visiting <__main__.Node object at 0x7fca2a0400b8>, containing data 7
Visiting <__main__.Node object at 0x7fca2a040cc0>, containing data 9
Visiting <__main__.Node object at 0x7fca2a040668>, containing data 9
Visiting <__main__.Node object at 0x7fca2a0406d8>, containing data 3
Visiting <__main__.Node object at 0x7fca2a040320>, containing data 4
Visiting <__main__.Node object at 0x7fca2a0406a0>, containing data 1
Visiting <__main__.Node object at 0x7fca2a040c88>, containing data 0
Visiting <__main__.Node object at 0x7fca2a040d68>, containing data 3
Visiting <__main__.Node object at 0x7fca2a040d30>, containing data 3
Visiting <__main__.Node object at 0x7fca2a040cf8>, containing data 4
```

If we were to number the nodes in the `example_tree` in the order that they were visited (and their `data` values printed), we would obtain the following figure.



PREORDER AND POSTORDER The above order of visiting (and printing) the nodes of the tree is commonly known as *preorder* traversal, which can equivalently be described as that traversal that processes (i.e., in this case, prints a message) the root node of the tree before it processes its subtrees.

The dual of preorder is *postorder* traversal, which processes the subtrees before it processes the root of the tree. As expected, the code for postorder traversal amounts to a simple exchange of lines 3–4 with the lines 5–6 in the code snippet of `df_visit` above. To keep things unambiguous, here are the two traversals again, except now, instead of printing the `data` attribute of the trees, we will simply store the subtree in a list. Both functions in effect return a list of subtrees of the given tree, ordered as they would be visited in preorder or postorder traversal.

```
1  def preorder(root):
2      """Get the subtrees in preorder of the tree with the given root"""
3      subts = [root]
4      for kid in root.kids.values():
5          subts.extend(preorder(kid))
6      return subts
7
8  def postorder(root):
9      """Get the subtrees in postorder of the tree with the given root"""
10     subts = []
11     for kid in root.kids.values():
12         subts.extend(postorder(kid))
13     subts.append(root)
14     return subts
```

```
>>> [tr.data for tr in preorder(example_tree)]
[5, 5, 3, 1, 7, 9, 9, 3, 4, 1, 0, 3, 3, 4]
>>> [tr.data for tr in postorder(example_tree)]
[3, 1, 5, 7, 3, 4, 9, 1, 0, 9, 4, 3, 3, 5]
```

Note that preorder is not the same as the reverse of the postorder. Now, since we did not specify any particular ordering of the subtrees of a given tree, in some cases it is indeed possible that preorder and postorder will be the reverse of each other. The order of subtrees is important in *search trees* that maintain some additional ordering invariants, so for such trees preorder and reverse postorder will be different.

4.2 SPECIAL CASE: BINARY SEARCH TREES

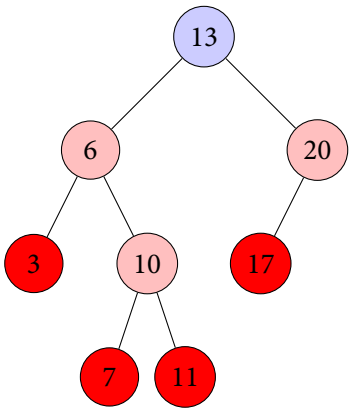
A special kind of search tree is a *binary search tree* (BST) where every internal node can have at most two elements, and where the subtrees of a node are sorted in such a way that the data elements of the left subtree are smaller than (<=) that of the root, which is in turn smaller than the data elements of the right subtree. We will use the following simplification of the general `Node` class for BSTs.

```
1  class BSTNode:
2      def __init__(self, data, left=None, right=None):
3          self.data = data
4          self.left = left
5          self.right = right
```

DEPTH-FIRST SEARCH A standard thing to do in a binary search tree is to find if a given *key* is present or not. The standard search algorithm is a depth-first search algorithm: to "visit" the current root, we compare the searched for key against the data stored at the root. If the data is found, we are done; otherwise, we go left or right depending on whether the key is smaller or larger than the data stored in the root. If we ever end up in the empty node, represented by `None`, then the key is not found. In Python:

```
def df_search(root, key):
    """Return True if key is in the tree with the given root, False
    otherwise"""
    # "visit" root
    if root is None: return False
    if key == root.data: return True
    # "visit" subtrees
    if key < root.data:
        return df_search(root.left, key)
    return df_search(root.right, key)
```

For example, consider the following BST.



Here is how `df_search()` behaves on it.

```
>>> example_bst = \
... BSTNode(13,
...     BSTNode(6,
...         BSTNode(3),
...         BSTNode(10, BSTNode(7), BSTNode(11))),
...     BSTNode(20, left=BSTNode(17)))
...
>>> df_search(example_bst, 7)
True
>>> df_search(example_bst, 42)
False
```

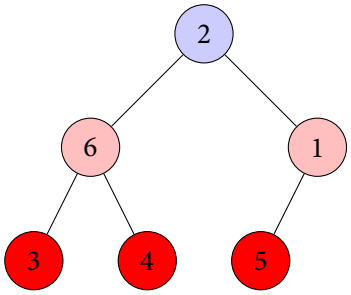
IN-ORDER TRAVERSAL For binary search trees, we not only have preorder and postorder traversals but also *inorder* traversal that visits the nodes of the left subtree first, then the current root node, and finally the nodes of the right subtree. It is pretty straightforward to adapt our earlier functions to define a recursive `inorder()` traversal.

```
def inorder(root):
    """Get the subtrees in order of the tree with the given root"""
    if root is None: return []
    return inorder(root.left) + [root] + inorder(root.right)
```

```
>>> [tr.data for tr in inorder(example_bst)]
[3, 6, 7, 10, 11, 13, 17, 20]
```

Keep in mind that in-order traversal has nothing whatsoever to do with the order relationships between the nodes in the tree. The tree need not obey the sortedness condition (left subtree below root

below right subtree) for in-order traversal to work. As an example, take this tree:



The in-order traversal of this tree would yield the nodes corresponding to the data in the tree in the following order: $[3, 6, 4, 2, 5, 1]$.

4.3 WORKLISTS

The depth-first search algorithms of the previous two sections were recursive. Because Python restricts recursion to a maximum of 1000 nested calls (by default), these recursive algorithms will not be able to traverse a tree with a depth greater than 1000. For this reason it is important to be able to recast the recursive algorithm into an *iterative* form, which does not run into the issue with recursion limits. The iterative version of the algorithm will just consist of a single `while` loop.

The broad structure of all iterative traversal algorithms is the same: they are based on *worklists*. A worklist is an abstract data structure that supports the following operations:

1. *Create* a worklist from an explicit list
2. *Test* if a worklist is empty
3. *Add* a new item to a worklist
4. *Extract* an existing item from the worklist, i.e., delete it from the list and return it.

We show this abstract traversal function using the functions `wl_create_singleton()` (to create a new worklist), `wl_is_empty()` (to test for emptiness), `wl_add()` (to add a new element), and `wl_extract()` (to extract an existing element).

```
1  def traverse_iterative(root):
2      """Iterative traversal of the tree with the given root"""
3      traversal_order = []                    # nodes in the order they are visited
4      worklist = wl_create_singleton(root)    # the worklist
5      while not wl_is_empty(worklist):
6          node = wl_extract(worklist)         # get the next element in the worklist
7          traversal_order.append(node)        # "visit" the node
8          for kid in node.kids.values():
9              wl_add(worklist, kid)           # add the descendant to the worklist
10     return traversal_order
```

As long as the given tree has finitely many descendants, this function will eventually run out of subtrees to add to the worklist in line 9, so the `while` loop started at node 5 will eventually end.

ITERATIVE DEPTH-FIRST TRAVERSAL Obviously one way to implement this abstract worklist data structure would be to use lists directly. Here is how we would implement the worklist functions:

```
wl_create_singleton(x):    [x]
       wl_is_empty(wl):    len(wl) == 0 (or just not wl in a conditional)
        wl_extract(wl):    wl.pop()
         wl_add(wl, x):    wl.append(x)
```

This yields the following implementation of iterative depth-first traversal.

```
1  def df_iterative(root):
2      """Iterative depth-first traversal of the tree with the given root"""
3      traversal_order = []                  # nodes in the order they are visited
4      worklist = [root]                     # the worklist
5      while worklist:
6          node = worklist.pop()             # get the next element in the worklist
7          traversal_order.append(node)      # "visit" the node
8          for kid in reversed(node.kids.values()):
9              worklist.append(kid)          # add the descendant to the worklist
10     return traversal_order
```

The way to think about this implementation is that the moment a node is removed from the end of the worklist with `.pop()`, it is replaced by its immediate descendants at the end of the worklist. Therefore, in the next iteration of the loop, the next node to be looked at will be one of the descendants of the node that was removed in the previous step. This mode of usage of a list is known as a *stack* or a *last-in-first-out* (LIFO) queue; think of it like a pile of playing cards, where the next card to be drawn is at the top of the pile, and you add cards by adding them to the top of the pile.

You may wonder why we used `reversed()` in line 8. This is because the first node we want to visit after visiting the given `node` is its first descendant *in the same order as* that returned by `.values()`. Since `.pop()` always removes the last item that was added, we need to make sure that the first item of `.values()` is the last item added to the worklist, i.e., that we add the `kids` to the worklist in the reverse order. Can we do postorder traversal in this iterative manner? Yes, and it is trivial: just drop the call to `reversed()` in line 8. However, the `traversal_order` will contain the reverse of the postorder (why?), so we need to change line 10 to read:

```
    return reversed(traversal_order)
```

ITERATIVE BREADTH-FIRST TRAVERSAL As mentioned, we can implement worklists in other ways. One way is to use a *first-in-first-out* (FIFO) queue, where the item in the worklist that was inserted the earliest is the next item to be extracted. FIFO queues are the kinds of queues you are used to from your daily life, such as the queue at a shop or at the bank.

In Python, you can still use lists to implement FIFO queues, where instead of `.append()` you would use `.insert()` to add the new items to the start of the queue, at the opposite end of where `.pop()` removes items from. However, it turns out that this is rather inefficient, since inserting in the front of a list causes the entire rest of the list to be copied one place over.[1]

Fortunately, there is an efficient queue data structure available as `collections.deque` (*deque* stands for ***d**ouble-**e**nded **que**ue*, which can rhyme with *deck* or *deek* as you prefer). Here is how the worklist looks like implemented as deques.

[1] Remember that Python implements lists using dynamically resizing arrays.

```
wl_create_singleton(x):    deque([x])
       wl_is_empty(wl):    len(wl) == 0 (or just not wl in a conditional)
        wl_extract(wl):    wl.popleft()
         wl_add(wl, x):    wl.append(x)
```

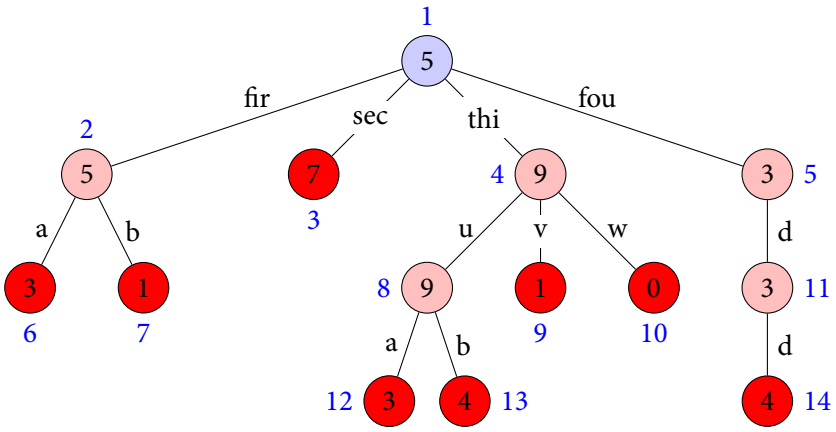
We thus obtain the following traversal, which is known as *breadth-first traversal*.

```
1  from collections import deque
2
3  def bf_iterative(root):
4      """Iterative breadth-first traversal of the tree with the given root"""
5      traversal_order = []                    # nodes in the order they are visited
6      worklist = deque([root])                # the worklist
7      while worklist:
8          node = worklist.popleft()           # get the next element in the worklist
9          traversal_order.append(node)        # "visit" the node
10         for kid in node.kids.values():
11             worklist.append(kid)            # add the descendant to the worklist
12     return traversal_order
```

What does this traversal do? Here is what we observe:

```
>>> [tr.data for tr in bf_iterative(example_tree)]
[5, 5, 7, 9, 3, 3, 1, 9, 1, 0, 3, 3, 4, 4]
```

If we were to number the nodes that are produced, this is what we would see.



As should be obvious from the figure, the traversal visits all the nodes at depth 0, then all the nodes at depth 1, then those at depth 2, and so on. Before starting nodes at depth $k+1$, the traversal will first have visited all the nodes at depth k and shallower. Why is this the case? Briefly, the way to interpret the worklist in this implementation is that it is always of the form

$$\texttt{deque}([n_1^k, n_2^k, \ldots, n_u^k, \quad d_1^{k+1}, d_2^{k+1}, \ldots, d_v^{k+1}])$$

where the n_i^k are nodes at depth k in the tree, and d_j^{k+1} are nodes at depth $k+1$ (for suitable i and j). When the algorithm hits line 8, it removes the node n_1^k from the left of the deque, and adds its descendants, say $d_{v+1}^{k+1}, \ldots, d_{v+w}^{k+1}$ to the right of the deque in lines 10–11, so the deque now looks like:

$$\texttt{deque}([n_2^k, \ldots, n_u^k, \quad d_1^{k+1}, d_2^{k+1}, \ldots, d_v^{k+1}, d_{v+1}^{k+1}, \ldots, d_{v+w}^{k+1}]).$$

Hence, we see that worklist always consists of some nodes at depth k, and some at $k+1$. Eventually the k-depth nodes will all be removed and the deque will be all $(k+1)$-depth nodes. At that point we will

shift perspective and consider the deque to be made up of an initial segment of $(k+1)$-depth nodes, and a trailing segment of $(k+2)$-depth nodes. Hence, all k-depth nodes are removed, in left to right order, before the $(k+1)$-depth nodes in left to right order, and so on. (See Exercise 2.)

Comparing depth-first and breadth-first traversal Consider the search problem: how to find a node in the tree by starting from the root. Depth-first search (i.e., search using depth-first traversal) would examine an entire subtree before it considers sibling subtrees: if it chooses the order in which to visit the subtrees in an optimized way, it can quickly hone in on the target node. This is particularly the case with binary search trees, where the search only needs to descend down to target node once, and hence the search has time complexity linear in the depth of the tree. However, if it is not easy to narrow down the search space efficiently, then depth first search may spend a lot of time exploring dead subtrees before it stumbles into the correct branch.

Breadth-first search, on the other hand, explores the tree level by level starting from the root. If it finds a target node, it will find one that is closest to the root of the tree, which is sometimes stated as breadth-first finding the *shortest solution* first. If the search problem is completely unconstrained, then breadth-first search will avoid getting stuck in deep but dead subtrees. However, breadth-first search cannot exploit any knowledge about the search space such as an idea of which subtrees are more likely to contain a target node. Moreover, since the number of nodes at the kth level of a tree is of the order of $O(2^k)$ (or $O(b^k)$ where b is the *branching factor* or the maximum number of children a node can have), it is clear that the time and memory demands of breadth-first search grows exponentially with the depth of the target node.

Combining the benefits: iterative deepening The above section may lead you to think that there are fundamental trade offs involved in selecting depth-first or breadth-first traversal. However, in practice the benefits of both traversals can be largely achieved without significant loss of performance by a technique known as *iterative deepening*, sometimes also known as *progressive deepening*.

Iterative deepening traversal is a refinement of the depth-first traversal technique, but it imposes a *depth bound* on the traversal to prevent the traversal from getting stuck in deep subtrees. Traversal is based on three scenarios with such bounds.

1. If a target node is found, the search simply succeeds.
2. When there are no more nodes in the worklist, the search either fails outright (if the depth bound was never hit), or fails for the given depth bound, because scenario 3 below is also triggered.
3. If an attempt is made to descend past the depth bound, the descendants of the node are silently discarded instead of being added to the worklist, but the fact that the depth bound was hit is remembered for scenario 2 above.

Globally, to maintain the invariant that the traversal will eventually find a target node if one exists in the tree, there is a master outer loop that keeps incrementing the depth bound every time case 2 above is triggered and then restarting the search. Hence the name of the algorithm: *iterative deepening*.

To see it in action, we are going to modify our `df_iterative()` from earlier. We call this variant of the function `id_dfs()` which takes an additional `test_fun` parameter which is a one-argument function that will return `True` if a given node is a target node. The function is shown in Figure 4.1. The depth bound is tracked using the variable `depth_bound`, which is incremented (line 37) every time a full DFS of the tree up to the current bound is completed. Note that if the bound is hit during traversal, this fact is just recorded in a Boolean variable `bound_triggered` (line 21); the node itself is not looked at in this case, and the overall DFS is not aborted. After all, a target node may be found somewhere else in the worklist.

```python
def id_dfs(root, test_fun):
    """Iterative deepening (iterative) depth-first search of the
    tree with the given root, where target nodes are identified
    by calling test_fun on them"""

    # The initial depth bound, which can be any natural number
    depth_bound = 0
    bound_triggered = True           # detect if the depth bound was hit

    while bound_triggered:           # begin by assuming the bound was triggered
        bound_triggered = False      # will be reset to True if triggered

        # Now we do a normal DFS, but the worklist keeps a pair of the
        # node together with its depth
        worklist = [(0, root)]
        while worklist:
            (depth, node) = worklist.pop()
            # check the bound
            if depth > depth_bound:
                # Record that the bound was hit
                bound_triggered = True
                # And keep going; forgetting about the node
                continue
            # At this point we know that we have a candidate node within bounds
            if test_fun(node):
                # The node was a target node, so just succeed
                return node
            # Otherwise, we add the descendants of the node to the worklist
            # by remembering to increment the depth
            for kid in node.kids.values():
                worklist.append((depth + 1, kid))

        # If we reach here, then we have done a full traversal of the part
        # of the tree that was not cut off by the depth_bound, so we need to
        # increment the bound (regardless of whether it was triggered or not)
        # so that more of the tree can be looked at.
        depth_bound += 1

    # If we reach here, then we have exhaustively checked the whole tree
    # (because the bound wasn't triggered, ending the while loop)
    # and we have failed to find a target node
    return None
```

Figure 4.1: Iterative deepening variant of depth-first search

4.4 APPLICATION: GAME TREES

The trees that we have been working with so far have been explicit objects created in memory as instances of the `Node` class (or `BSTNode` in the case of BSTs). Due to the limitations of reasonable computers, such tree objects will not only be finite but also tend to be fairly small. However, often we need to search in trees that are far too large to ever be fully computed, or even infinite in size. As an example, we will consider *game trees*, which are an abstraction for representing the tree of *game states* that arise in games with discrete moves and outcomes.

There are many kinds of games and hence many kinds of game trees. In this section we will focus on *deterministic two-player competitive games*, in which two players play on a shared *board* or *arena* with the goal of winning or defeating their opponents. Some well known examples of such games include Chess, Go, Backgammon, and Shogi. (Limiting ourselves to deterministic games simplifies the setting somewhat, but excludes many interesting options including card games and dice games.)

Games such as Chess and Go are known to have very large *search spaces*. Take Chess, for instance. Whenever a player has to move, they have the option of moving any of their pieces on the board to any legal position in the board. At the start of the game this gives the opening player 20 possible moves (8 pawns, each potentially moving one or two spaces, plus the two knights each potentially moving to two possible spots). The opponent in turn has 20 possible responses. Eventually, as the pieces move about more moves become possible, while later in the game the players will have lost a lot of pieces and have fewer movement possibilities.

GAME TREES ARE IMPLICIT TREES It is convenient to think of all the possible games of Chess as being laid out in a *game tree*. Think of each configuration of the Chess board as a node in a hypothetical tree, with the root being the starting Chess board before any moves have been made. The descendants of the root node are the board states that result for all the moves of the opening player, then *these* nodes have descendants based on the responses of the second player, and so on, alternating players at every level of the tree.

The full Chess game tree is estimated to have about $35^{80} \approx 10^{120}$ possible board states, which is likely impossible to store using the combined matter of the observable universe (which is estimated to have only about 10^{82} atoms). Games like Go have even more states (19×19 Go has 2.08×10^{170} possible boards). It is utterly infeasible to even compute, say, the full game tree for *five* pairs of Chess moves ($\approx 8.3 \times 10^{55}$ boards). Therefore, for such a situation, the game tree is not represented as a full tree in memory, but rather in terms of a tree that is computed on the fly. More precisely, one would define a class `Board` to represent a state of the chessboard, which would look something like this:

```
class Board:
    def __init__(self):
        # set up the initial board

    def legal_moves(self):
        # return a list or a generator over the valid moves

    def make_move(self, move):
        # return a *new* board that results from playing the
        # given move on the current board (self)
```

Then, in the `df_iterative()` algorithm or its DFS variant `id_dfs()`, in the case where we iterate over the children of the current node of the tree being traversed, we would instead iterate over the legal

moves on the current board produced by the `.legal_moves()` member function. If we actually ever need to consider playing the move, we would use the `.make_move()` member function to get a new instance of `Board` representing the board state that would result from making that move.

Iterative deepening with heuristics As already mentioned, it is not possible to keep even a few levels of the entire game tree of Chess or Go in memory. Therefore, when a computer plays these games, they play with *bounded lookahead*, which means that they explore as far as they can in the game tree under the memory and time constraints, and then they guess whether the game states are favorable to them or to their opponent. This guesswork is called a *heuristic evaluation*, and it is intended to look at a *single* board state and arrive at a measure of its *goodness* for the corresponding player.

Coming up with good heuristics for particular games is a delicate art. Human players train all their lives to build good heuristic functions in their brain: the best human players at a game are able to make more accurate heuristic assessments over more boards faster than other players, allowing them to look ahead further in the game tree than their opponents. A computer will likewise either need to figure out the heuristic function by exhaustive training and *machine learning*, or be programmed to employ heuristics that have already been determined to be good.

Practically, a heuristic function is a function of two arguments: the first argument is the current board being evaluated, and the second is whose turn it is to play on the board. The players on the board will be numbered 0 and 1, with the computer always choosing to be 0. Thus, the heuristic function in the computer will try to evaluate the board from the perspective to getting player 0 to win. There are three possibilities for the heuristic function:

1. If the board represents a certain victory for 0, then the heuristic value is generally taken to be $+\infty$ (approximated in Python with `math.inf`). This is the best possible outcome for player 0, and therefore the search algorithm will try to force the game to evolve towards one such board. Note that this certain victory condition does not necessarily mean that the board represents an already achieved victory condition for player 0: rather, it encapsulates the condition that no matter whose turn it is to play and what moves that player makes, the outcome will be a certain victory for player 0.
2. Dually, there are boards that represent certain defeat for player 0, and for these boards the heuristic evaluates to $-\infty$. The computer will try its best never to end up in one of these board states.
3. The vast majority of boards, however, will have an unclear victor, in which case the heuristic function returns a number that is indicative of how likely it is for player 0 to win with the given board. If there are more ways to win from the board, its heuristic score should be higher, and likewise if there are more ways to lose then the score should be lower.

Given this setup, we can modify our `id_dfs()` algorithm to work as follows:

- While we are under the depth bound, we would *expand* the game tree by computing the successor boards of the current board.
- If we hit the depth bound, instead of discarding the board we perform a heuristic evaluation of the board instead.
- Eventually the algorithm should try to find the *board with the best heuristic rating within the depth bound*.

This board is where the computer will try to direct the play. However, note that it is not enough to simply find the best possible board for player 0: in order to force the play into that board, it must also be the case that the opponent (player 1) is not able to win from intermediate stages in the path to that board. This requires a non-trivial modification to the DFS algorithm.

MINIMAX The variant of DFS that the computer uses to search for best plays is generally known as the *minimax* algorithm (sometimes also called *minmax*). The goal of the algorithm is to find the best possible *next move* from the root of the game tree, which it performs by means of *evaluating* the root board.

1. If the root board is already at the depth bound, then instead of computing its successor boards, a heuristic evaluation is performed instead. This heuristic evaluation *stands in* for its true evaluation (which would require exploring beyond the search bounds).
2. Otherwise, the board is expanded and an evaluation is performed recursively on all the successors. If the current turn is player 0's, then the evaluation of the root board is the *max* of the evaluations of all the successor boards. Otherwise, if the current turn is player 1's, then the evaluation of the root board is the *min* of the evaluations of all the successor boards.

Remember that the evaluation always computes the goodness of the board from the perspective of player 0. On player 0's turn, their goal is to make a move that maximizes the goodness of the board that results from the move: hence, they play a *maximizing move*, which is a move that maximizes the goodness. Dually, on player 1's turn, this player wants to play a move that makes life as hard as possible for player 0, and so they play a *minimizing move*, which is a move that minimizes the goodness of the resulting board for player 0. Because of this alternation of maximizing and minimizing moves, player 0 is often called the *maximizing player* and player 1 is the *minimizing player*.

EXAMPLE: TIC-TAC-TOE As every one knows, the game of *tic-tac-toe* (also known as *crosses and noughts*) is played on a square grid where the players alternate in making X and O marks in unoccupied cells of the grid. The first player to make a line of their corresponding sigil that spans an entire row, column, or diagonal, wins the game.

Let us take the perspective that the maximizing player is X. The leaf states of the tic-tac-toe game will have the following evaluations, depicted below together with an example board.

X-win: evaluation of $+\infty$	O-win: evaluation of $-\infty$	Tie: evaluation of 0
X O X O X O X	X O X O X O	X O X X O O O X

Figure 4.2 shows a fraction of two game trees; in the first, X wins at the root of the tree, while in the second X loses. The evaluations are shown in pink boxes.

For the simple case of 3×3 tic-tac-toe boards, the full game tree is small enough that it can be precomputed and stored in any average computer's memory. However, even on 4×4 boards, a full minimax exploration of the board will encounter noticeable slowdown. So, for such boards we would like to use a heuristic evaluation with bounded lookahead. Specifically, as the minimax algorithm stipulates, when we hit the depth bound we would just examine the board *without making any moves* to determine the goodness of the board for the X player.

Definition 4.3 (Tic-Tac-Toe heuristic)**.** *The heuristic evaluation of an $n \times n$ tic-tac-toe board is computed by summing the following quantities:*

- $+\infty$ *if X has a guaranteed win*
- 10^i *for each row, column, or diagonal containing i X's and $n - i$ empty cells*
- -10^i *for each row, column, or diagonal containing i O's and $n - i$ empty cells*
- $-\infty$ *if O has a guaranteed win*

In any valid game it is impossible for both X and O to win, so there will never be a situation where $+\infty$ and $-\infty$ are being summed in the heuristic.

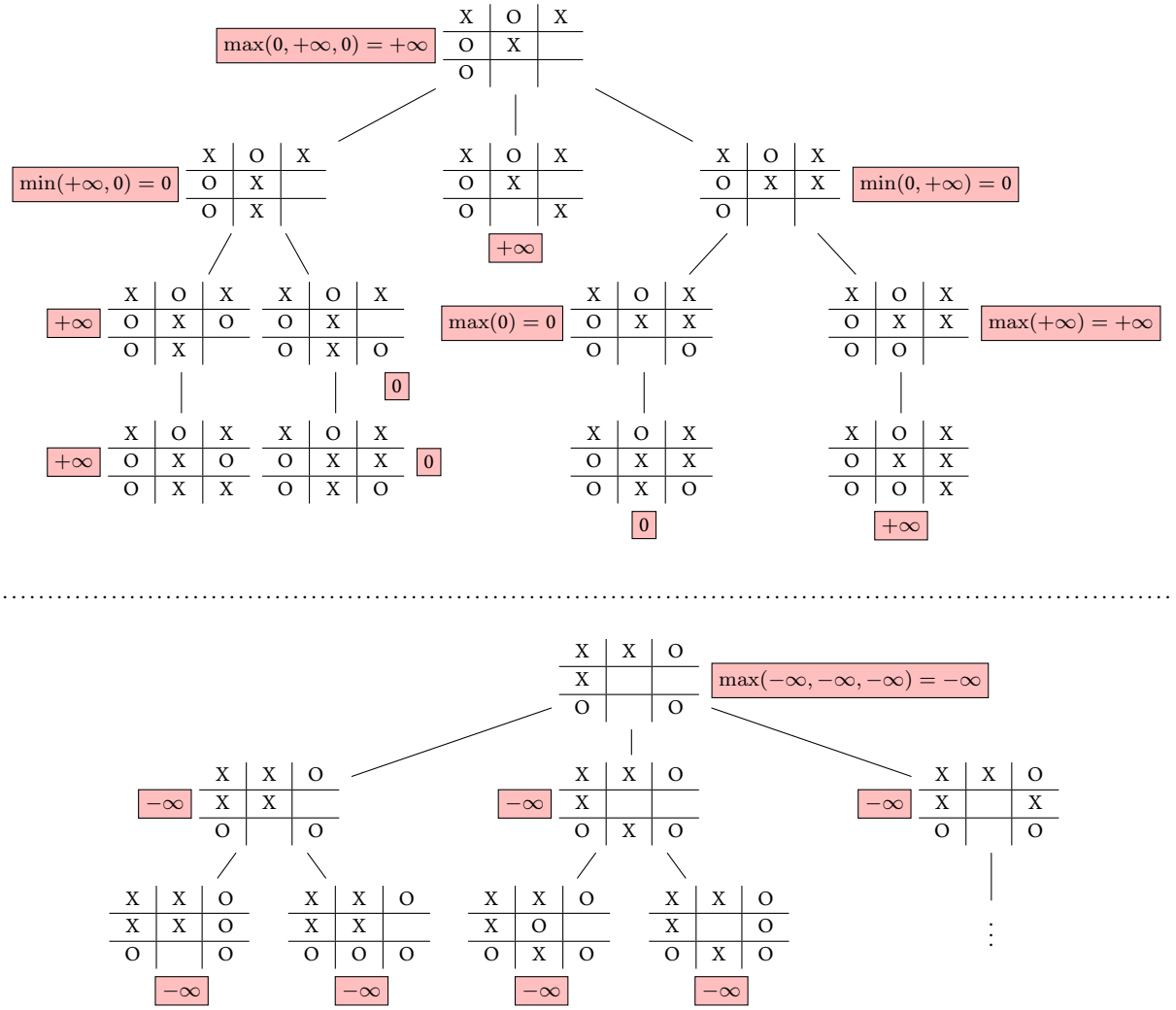


Figure 4.2: Two examples of 3 × 3 tic-tac-toe game trees. X wins in the first, and loses in the second. Observe that in the first game, X has only one maximizing move – to play in the bottom right corner. If X plays anywhere else, O will be able to bring the game to a draw.

As an example of the heuristic, take the following 4×4 tic-tac-toe board.

O	X	O	
		X	
O	X		

Its heuristic evaluation is:

$$\begin{aligned} & 1+1 && \text{(1st row, 4th col)} \\ & +10 && \text{(3rd row)} \\ & +(100+100) && \text{(2nd col, nwse diagonal)} \\ & -(1+1) && \text{(1st row, 4th col)} \\ & -(100+100) && \text{(1st col, nesw diagonal))} \\ & = +\ 10 \end{aligned}$$

4.5 EXERCISES

1. Write the function `breadth(root)` that returns the breadth (cf. Definition 4.2) of the tree rooted at `root`. Using preorder traversal, return a result in time $O(n^2)$ where n is the number of nodes in the tree. Can you do it in $O(n)$ time?

2. Refine the sketch of the argument about the correctness of breadth-first traversal at the end of Section 4.3 into a precise inductive proof of a corresponding correct theorem. The invariant you will need to drive your theorem is almost completely described in the sketch, but you need to pay more attention to exactly what measure is being inducted over.

5 LOCAL SEARCH AND OPTIMIZATION

In Chapter 4 you were introduced to several kinds of traversal techniques for trees, both trees that are constructed in the form of explicit data structures, and implicit trees that are generated on the fly as needed. In this chapter we will broaden our perspective from tree-like structures to a more general setting. We will consider the question of how to find the solution to a particular problem that can be described in *spatial* terms, where candidate answers are related to other candidate answers in a quantifiable manner. This relation can be a global ordering of all candidate answers in terms of a *cost* assignment, or it can be a relative ordering in the form of a *distance* function for pairs of candidate answers.

5.1 SIMPLE DIRECTED GRAPHS

KEY CONCEPTS To keep things concrete, we will consider *graphs*, which are a fundamental kind of data structure common to many areas of computer science.

Definition 5.1 (Graph). *A* graph *G is a structure $\langle V, E\rangle$ where:*

- *V is a set of* nodes *(also known as* vertices*). Nodes can contain additional data items, just like nodes in search trees. The node set V need not be finite, but in computer science we generally only consider* finite graphs *made up of a finite node set.*
- *$E \subseteq V \times V$, called the* edge relation*, defines when two nodes are connected. Elements of E are called* edges*. For any edge $(u, v) \in E$, we say that u is the* source *of the edge, and v is the* target *of the edge.*

To be clear, this definition defines *simple directed graphs*, sometimes called *digraphs*, where the edge relation is a collection of ordered pairs. There are a number of other kinds of graphs that we will not consider in this course, but briefly:

- A *undirected graph* is like a directed graph, except the edge relation is not ordered; instead, it is just a set of 2-element sets of nodes, $E \subseteq \{\{u, v\} : u, v \in V \text{ and } u \neq v\}$.
- A *multi-graph* is a graph where instead of the edge relation being a set, it is a *multi-set*, i.e., an unordered list. This means that there can be several edges between the same pair of nodes.
- A *hypergraph* is a graph where the edge relation connects not just two nodes but an arbitrary subset of nodes. The edges of a hypergraph are often visualized as *clouds*.

There are a number of related concepts on graphs that we list below for convenience.

Definition 5.2 (Graph concepts). *For a digraph $G = \langle V, E\rangle$:*

- *for any node $v \in V$, its* next set *is a subset of nodes, written as $next(v)$, defined as:*

$$next(v) = \{w \in V : (v, w) \in E\}.$$

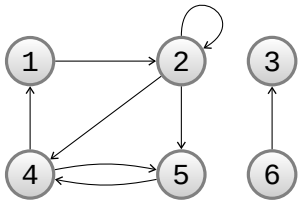
  Likewise, its prev set*, written $prev(v)$, is defined as:*

$$prev(v) = \{u \in V : (u, v) \in E\}.$$

- *The* in-degree *of a vertex $v \in V$, written as $indeg(v)$, is just the cardinality of its prev set, i.e., $|prev(v)|$. Likewise, the* out-degree *of $v \in V$, written as $outdeg(v)$, is $|next(v)|$.*
- *A sequence of nodes $v_0, \ldots, v_n \in V$ is said to be a* path *if for every $i \in 0..n-1$ it is the case that $(v_i, v_{i+1}) \in E$.*
- *Two nodes $u, v \in V$ are connected if there is path whose first node is u and whose last node is v.*

GRAPHS IN PYTHON There are many algorithms related to graphs that we will not cover in this course, since we are only using graphs to illustrate a concept. However, it is still important to know how to represent and operate on graphs. We will use a representation technique known as the *adjacency list* representation, which is designed to work with graphs where most pairs of nodes don't have an edge between them; such graphs are known as *sparse*. Trees are an example of sparse graphs.

To build up to the implementation of graphs in Python, let us begin with the simplest possible representation of the following graph.



Each node is drawn as a shaded circle with a unique integer label, and directed edges are drawn using arrows. In the adjacency list representation, we would represent this graph as a mapping (i.e., a dictionary) from the node labels to their next or prev sets. Let us, for instance, just record the next sets, which gives us the following dictionary.

```
>>> ex_graph = {1: {2}, 2: {2, 4, 5}, 3: set(), 4: {1, 5}, 5: {4}, 6: {3}}
```

For instance, the next set for the node labeled `2` is given by `ex_graph[2]`, which is the set `{2, 4, 5}`. To see an example of the use of such a graph structure, let us program a function that tests if two nodes in the graph are connected. To do this, we will use recursive depth-first traversal starting from the source node using the next sets to make recursive calls, and try see if we land on the target node. If we complete our exploration of the graph without finding the target node, then the two nodes are not connected. Here is the simple implementation.

```
1 def is_connected(graph, source, target):
2     """Return True if source and target are connected in graph; False otherwise"""
3     # print('source =', source)  # we will uncomment this later
4     if source == target:
5         return True
6     for n in graph[source]:      # iterate over the next set of source
7         if is_connected(graph, n, target):
8             return True
9     return False
```

```
>>> is_connected(ex_graph, 5, 2)
True
>>> is_connected(ex_graph, 6, 2)
False
```

Unfortunately, this implementation doesn't work for every pair of nodes. Imagine what happens on a call to `is_connected(ex_graph, 5, 3)`. The first recursive call will be with a new source of 4, which will itself make a recursive call with source 1, and then finally end up with a call with source 2. Here, we get stuck, because the first recursive call will be back to the case with source 2, and since there is no way to prevent a recursive call, this scenario will just keep repeating forever. We can see this in action if we uncomment line 3 so that it prints out the `source` on every call to `is_connected()`.

```
>>> is_connected(ex_graph, 5, 3)
source = 5
source = 4
source = 1
source = 2
source = 2
### above line repeats 995 more times before:
RecursionError: maximum recursion depth exceeded
```

To fix this issue, we need to build in some *cycle detection* into the algorithm so that when it returns to a node it has already started to traverse from earlier, which corresponds to an identical call to `is_connected()`, the algorithm will *abort* early. If the target is not on the path that leads back to a node seen earlier, then all the edges on the path can safely be removed from consideration. Furthermore, since we are traversing the graph in a depth-first manner, we know that if a path is not found in a recursive call to `is_connected()` we can simply remove all the nodes in the next set (and the edges they participate in) from the graph entirely, because (by induction) there is no path from any of the successor nodes to the target. This intuition is captured by means of a *seen set*, which is a set of nodes that have have all already been `source` arguments to some call to `is_connected()`. This set will *prune* repeated calls to `is_connected()` by returning `False` early. Here is the updated code.

```
1  def is_connected(graph, source, target, seen):
2      """Return True if source and target are connected in graph; False otherwise"""
3      print('source =', source)
4      if source in seen:
5          print('  cycle')
6          return False
7      seen.add(source)
8      if source == target:
9          return True
10     for n in graph[source]:  # iterate over the next set of source
11         if is_connected(graph, n, target, seen):
12             return True
13     return False
```

```
>>> is_connected(ex_graph, 5, 3, seen=set())
source = 5
source = 4
source = 1
source = 2
source = 2
  cycle              # {2, 2} cycle
source = 4
  cycle              # {4, 1, 2} cycle
source = 5
  cycle              # {5, 4, 1, 2} cycle
source = 5
  cycle              # {5, 4} cycle
False
```

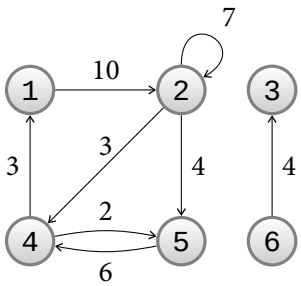
The output has been annotated slightly with a comment on the right identifying which cycle was detected. You should play with variations of this code to make sure that you understand why it works. For instance, do you see exactly why we annotated the cycles as we did above?

5.2 GREEDY SEARCH

Node and edge data In practice, both nodes and edges in a graph can contain additional data. Moreover, in our implementation of `is_connected` we traversed the graph edges in the direction of source to target, but we may sometimes want to go in the reverse direction: with the simple dictionary we had, it is difficult to efficiently go backwards, since we need to search the entire graph to compute the prev sets (do you see why?). For this reason it is customary to separate the node, next, and prev sets into three separate components that are kept in sync.

In fact, each of these sets is represented as a dictionary. Figure 5.1 shows the full implementation. The `._nodes` dictionary maps every node id (an `int` like before) to its corresponding data, the `._nxt` dictionary maps every source node to a target it is connected to together with the edge data, and the `._prv` dictionary maps every target node to the source nodes that connect to it together with the edge data. To add a node to the graph we would use `.add_node()` to extend the `._nodes` dictionary, while to add an edge we would extend *both* the `._nxt` and `._prv` dictionaries. We also override the `.__getitem__()` special function so that we can get the node data for node `n` from graph `g` by the form `g[n]`, while we can get the edge data for the edge from `u` to `v` as `g[u, v]`.

Let us use it to build the following graph, where the data stored at the nodes is trivial (all `None`, say), and where the edge data are positive natural numbers that denote a *cost* of traversing that edge.




```
>>> ex_graph = DiGraph(nodes=[(i, None) for i in range(1, 7)],
...                    edges=[(1, 2, 10), (2, 2, 7), (2, 4, 3), (2, 5, 4),
...                           (4, 1, 3), (4, 5, 2), (5, 4, 6), (6, 3, 4)])
>>> ex_graph[1, 2] # edge data for the (1, 2) edge
10
>>> ex_graph.is_edge(1, 5)
False
>>> for n in ex_graph.nxt(2): print(n) # print the next set for node 2
...
2
4
5
```

Finding low cost paths Consider the problem of lowering the cost of getting from one point to another. This problem can be solved in two drastically different methods. The first method is to consider the graph as a whole and try to find the best possible—lowest total cost—paths that connect any two nodes that have a path between them. These least cost paths will generally be in the form a collection of trees called the *spanning forest* of the graph, and there are algorithms that can find such spanning forests fairly quickly. You'll see such algorithms in CSE 103 and other courses.

Let us consider here the other perspective, which doesn't necessarily attempt to find the minimal path, but tries to make intelligent choices for building the paths in a node by node basis. Such a perspective is called *local search*, because the search algorithm only looks at a small portion of the graph at any given

```
class DiGraph:
    def __init__(self, nodes=None, edges=None):
        self._nodes = dict()
        self._prv = dict()
        self._nxt = dict()

        # populate the graph based on the nodes and edges parameters
        if nodes is not None:
            for (v, data) in nodes:
                self.add_node(v, data)
        if edges is not None:
            for (u, v, data) in edges:
                self.add_edge(u, v, data)

    def add_node(self, v, node_data):
        """Extend the graph with a new node v with given node data"""
        assert isinstance(v, int)
        assert v not in self._nodes
        self._nodes[v] = node_data
        self._nxt[v] = dict()
        self._prv[v] = dict()

    def add_edge(self, source, target, edge_data):
         """Add an edge between source and target nodes"""
         assert source in self._nodes
         assert target in self._nodes
         self._nxt[source][target] = edge_data
         self._prv[target][source] = edge_data

    def is_edge(self, source, target):
        """Is there an edge from source to target?"""
        return target in self._nxt[source]

    def __getitem__(self, key):
         """Access node data with accessor [id] and edge data
         with accessor [source_id, target_id]"""
         if isinstance(key, int):
             return self._nodes[key]
         elif isinstance(key, tuple):
             (source, target) = key
             return self._nxt[source][target]
         else: raise ValueError

    def nxt(self, v):
        """Iterator over the next set of v"""
        assert v in self._nodes
        return self._nxt[v].keys()

    def prv(self, v):
        """Iterator over the prev set of v"""
        assert v in self._nodes
        return self._prv[v].keys()
```

Figure 5.1: Simple directed graph implementation with node and edge data

moment. Like with game trees earlier, this kind of search is compatible with data structures that are too big to represent physically in a computer, or possibly even infinite.

The local search algorithm we will consider here is known as *greedy search*. It works as follows: whenever we make a recursive call in a depth-first algorithm such as `is_connected()`, the order in which we consider the successor nodes in the next set will be in increasing order of the costs to reach that node. So, we will pick the lowest cost edge first, and only if that does not lead to a path will we consider the edge with the next higher cost. In Python it is very easy to sort the nodes in this fashion: recall that we already have an iterator over the next set in the form of the `.nxt()` function. Any finite iterator can be turned into a *sorted iterator* by means of the built in **`sorted`**`()` function of Python; moreover, like the **`min`**`()` and **`max`**`()` functions, the **`sorted`**`()` function takes a `key` argument that is a function that is applied to the elements of the underlying iterator when consider their relative ordering. Here is how we compute the next "set" of the node 2 sorted by the costs:

```
>>> sorted(ex_graph.nxt(2), key=lambda n: ex_graph[2, n])
[4, 2, 5]
```

We can now update our `is_connected()` function to return not just whether the source and target nodes are connected but also an upper bound in the minimal cost path between them by means of greedy search. Let's call this function `greedy_connected()` to keep it distinct from `is_connected()`.

```
1  def greedy_connected(graph, source, target, seen):
2      """Return an upper bound on the min cost path between source and target
3      if they are connected, and None if they are not connected"""
4      if source in seen:
5          return None             # we still reject cycles
6      seen.add(source)
7      if source == target:
8          return 0                # the distance from a node to itself is 0
9      for n in sorted(graph.nxt(source), key=lambda n: graph[source, n]):
10         # cost to reach target from n
11         cost = greedy_connected(graph, n, target, seen)
12         if cost is not None:
13             # we add to it the cost to reach n from source
14             return cost + graph[source, n]
15     return None
```

```
>>> greedy_connected(ex_graph, 2, 5, seen=set())
5
```

Note that there is a path of a shorter cost from node 2 to node 5, which is the direct edge of cost 4. Unfortunately, greedy local search considers the immediate edge of cost 3 out of node 2 first, which does lead to a successful path and hence the direct edge is not even considered. This is why we say that it computes an upper bound on the least cost path. That is, it returns a cost that it can guarantee, but there may be lower cost paths it hasn't explored.

5.3 OPTIMIZATION PROBLEMS: HILL CLIMBING AND GRADIENT DESCENT

The greedy search we saw in the previous section is an example of a class of algorithms known as *local search* algorithms, and such algorithms are generally considered in the branch of computer science known as *optimization problems*.

Definition 5.3 (Optimization problem). *Given a non-empty set S and an objective function $cost : S \to \mathbb{R}^+$, the* optimization problem *is to compute its global minimum, i.e., $\operatorname{argmin}_{x \in S} cost(x)$.*

Note that this is a different sense of "optimization" than what you may be used to: we are not trying to make existing code faster, which is often (and usually erroneously) referred to as *optimization*.

An optimization algorithm based on local search tries to find the best parameter for the given objective function by starting from an initial guess of the parameter and then iteratively refining the parameter little by little until the value of the objective function cannot be improved any more. When that happens, if the parameter is a *local* minimum or maximum for the objective function.

Definition 5.4 (Local search). *Let a non-empty set S and an objective function $cost : S \to \mathbb{R}^+$ be given. A* local search *algorithm constructs a* neighborhood function $B : S \to \mathcal{P}(S)$[1] *and a sequence of elements $x_0, x_1, x_2, \ldots \in S$ where $x_0 \in S$ is chosen by some (unspecified) means*[2] *and*

$$x_{i+1} = \operatorname*{argmin}_{x \in B(x_i)}(cost(x) - cost(x_i)) \quad \text{for all } i \in \mathbb{N}.$$

The sequence is terminated with x_k when either $B(x_k) = \emptyset$ or $|cost(x_{k+1}) - cost(x_k)| < \varepsilon$ for some fixed $\varepsilon \in \mathbb{R}^+ \setminus \{0\}$. This last value of the sequence is the output of the algorithm.

Ideally, a local search algorithm will have the following properties:

1. It will indeed solve the optimization problem, i.e., the algorithm should not settle on a local minimum that is not also a global minimum. One way to guarantee this is that the neighborhood function doesn't have too "local" a perspective, so that it can indeed see neighboring minima.
2. It will find a solution quickly. We want to minimize the number of iterations of the optimization loop—the number of elements of the sequence computed—which means that we should select the neighborhood function to take large steps. This sets up a fundamental trade-off: if B picks elements that are too distant from its input, then the inner loop of the local search algorithm can oscillate. Also, if B contains too many elements, then the search tree for local search can get too big to compute the $argmin$ in the inner loop of the search efficiently.

To address these constraints, it is common to allow the neighborhood function to itself evolve as the initial approximation keeps being refined. The search algorithm can, for instance, use a sequence of neighborhood functions $(B_i)_{i \in \mathbb{N}} : S \to \mathcal{P}(S)$ and:

$$x_{i+1} = \operatorname{argmin}_{x \in B_i(x_i)}(cost(x) - cost(x_i)) \quad \text{for all } i \in \mathbb{N}.$$

GRADIENT DESCENT If the parameter set S is dense and the objective function $cost$ is continuous and differentiable, then one way to define the neighborhood functions for local search is to pick all the points a given *step size* away along its negated *gradient*. Recall that, seen as a vector, that the gradient always points in a direction of maximal increase in the objective function; thus, if the objective function is continuous and differentiable, the gradient will always unerringly point towards a local maximum. Nevertheless, if the step size is too large, then the neighborhood will overshoot the local minimum, leading to oscillation. Thus, generally speaking the step size will be proportional to the magnitude of the gradient.

From the perspective of computer science, gradient descent is usually an idealized version of what can actually be implemented with finite precision floating point numbers. Usually, as the approximation

[1] $\mathcal{P}(-)$ is the powerset operator, i.e., $\mathcal{P}(S)$ is the set of subsets of S.

[2] For example, randomly.

nears a local minimum, the gradient starts yielding updates to the approximation that are too close, causing floating point underflows. When a floating point number is too close to zero, ordinary arithmetic operations on it can introduce arbitrary errors; if this is not accounted for, the local search algorithm may spin the approximation off chaotically just as it approaches a local minimum. To avoid this we can use arbitrary precision numerical libraries (as provided by, for instance, the `numpy` package), which are slower than finite precision floating point numbers implemented in hardware, or we can cut off the search when the magnitude of the gradient is too small (although even detecting when the magnitude is too small can trigger underflows).

Hill climbing When the parameter space S is discrete, it may not be possible to determine a suitable notion of a gradient. In such situations, the neighborhood function is not interpreted mathematically but algorithmically. In order for the local search algorithm to compute a result, it is enough that $x \in B(x)$ for every point $x \in S$. Moreover, instead of updating the approximation to one that lowers the cost as much as possible in its neighborhood, we can simply pick *some* point of lower cost, not necessarily the point that decreases the cost the most. The resulting algorithm often goes by the name *hill climbing*, which usually is also defined to find *maxima* rather than *minima*. Gradient descent is just a version of hill climbing where we take the *steepest ascent* (in the opposite direction) in the neighborhood.

Escaping local minima: Random Restart and Annealing All local search algorithms are highly sensitive to the choice of the neighborhood function. If the neighborhood is too small, the algorithm will tend to get stuck in local extrema, but if it is too big then search may oscillate between extrema or just circle around a given extremum. Two general techniques have been proposed for mitigating these negative tendencies of local search: *random restart* and *annealing*.

In the *random restart* approach, the neighborhoods are kept small in the vast majority of the steps of the search loop, but every once in a while (randomly, usually with an exponential distribution) a very large neighborhood is selected. This large step is called a "random restart", and the algorithm is considered to be done after it has had a fixed number of such restarts. Another alternative is to wait for the search to converge at the first extremum, and then to restart the search from scratch at another randomly chosen starting point x_0. Keep doing this as long as search keeps converging on better and better extrema, and stop when no better extrema are found.

The *annealing* approach mimics the physical process of cooling by modifying the step size by smaller and smaller values as the iteration continues. Initially the neighborhoods are large because of a large step size, which will hopefully allow the search to escape from localized hills and valleys. However, gradually the step size reduces (cools) so that eventually the search would settle in the best hill/valley it has found and converge to its apex/nadir.

5.4 EXAMPLE: THE N-QUEENS PROBLEM

To illustrate hill climbing and the random restart strategy for escaping local minima, we are going to use the example of the n-queens problem, which is the problem of placing n queens in an $n \times n$ chessboard in such a way that no two queens threaten each other. That is, every queen is alone on her row, column, and both diagonals. This problem can be shown to be solvable for any $n \in \mathbb{N}$ except $n = 2$ and $n = 3$ (cf. Exercise 2.) Examples of solutions for board $n = 4$, $n = 6$, and $n = 8$ are shown in Figure 5.2.

While it is possible to solve the 8-queens problem with standard divide-and-conquer backtracking search, that is close to the limit of what is solvable with a modern computer in a decent amount of time.

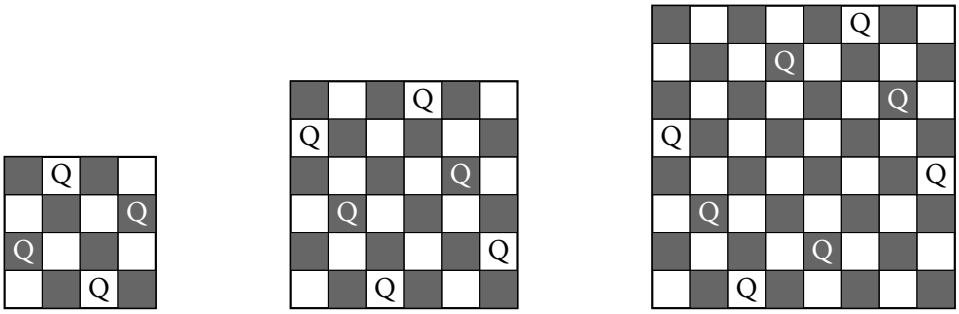


Figure 5.2: Some solutions for the n-queens problem for $n = 4$, $n = 6$, and $n = 8$.

Instead, we can set up a simple local search procedure to find a solution. The algorithm proceeds as follows:

1. Start with an initial candidate solution that places one per column of the board, with the rows that the queens are placed on chosen randomly.
2. If there are no queens under threat, we are done.
3. Otherwise, find one queen that is threatened and move it to a different row in its column so that it is not threatened. The neighborhoods of the candidate "solution" are all the boards that are reachable by moving a single queen in its column, meaning that each board potentially has 64 neighboring boards. Of them, we of course remove the boards where the moved queen remains under threat.
4. If there are threatened queens but no single queen can be moved to remove threats, then we have reached a local extremum without finding a solution. In this case we restart at step 1 by reshuffling the board. This is our random restart.

5.5 EXERCISES

1. Update the `is_connected()` and `greedy_connected()` functions from Sections 5.1 and 5.2 so that they also show the sequence of nodes traveled in the path that these functions find. For example:

```
>>> greedy_connected(ex_graph, 2, 5, seen=set())
([2, 4, 5], 5)  # 2-tuple of path and cost
```

2. Prove by mathematical induction that the n-queens problem can be solved for any $n \in \mathbb{N} \setminus \{2, 3\}$.

3. Implement the local search algorithm for the n-queens problem exactly as described at the end of Section 5.4.

4. Implement a variant of the n-queens local search algorithm as follows: we say that a queen is k-threatened if there are k other queens that threatened it. In the local search step, find a queen that is maximally threatened, and move it to such a place that its threat is strictly lowered, *while simultaneously ensuring that the maximum threat for all queens does not increase* (it can stay the same). This defines a slightly larger neighborhood for the candidate solutions, with more flexibility for finding solutions, so it is less likely to get stuck in local extrema.

5. Implement a variant of the previous exercise where we select a queen and a move for the queen that reduces the sum of all threats on the board as much as possible. This is a *steepest ascent* variant of the algorithm that is used by the best n-queens sol veers.

6 INTERACTIVITY

When writing programs that are intended to be used *interactively*, which is to say that the program receives, processes, and responds to user input, it is very important to be *responsive*. Studies have shown that most humans notice response delays as low as 1 ms and are irritated to the point of giving up by response delays of around 400 ms.[1] In this chapter we will explore some aspects of interactivity, specifically in the context of interactive computer graphics.

6.1 INTERACTIVE GRAPHICS: KEY CONCEPTS

A modern computer display is a 2-dimensional array of *pixels*: each pixel displays a single color at a given intensity, which is in turn determined by the intensities of its red, green, and blue components (called *channels*). For a long time each channel has been limited to 8 bits, so that each pixel can be completely described in 24 bits. With 32 bit words, this means that each pixel can be contained in a single word with 8 additional bits available for pixel-specific data such as transparency.[2] These pixel intensities are stored in a *framebuffer*, which is a shared area of memory that is written to by programs and read from by the computer's display hardware. With a display with $1920 \times 1080 = 2073600$ pixels, we would need $2073600 \times 4 \approx 7.91$ MB of memory dedicated to the framebuffer.

A snapshot of the contents of the framebuffer is called a *frame*. The display hardware displays a frame by synchronizing each pixel of the display to the intensities of the corresponding entry of the framebuffer. This process is called a *refresh*. The display hardware refreshes the display a fixed number of times per second, called its *refresh rate*, measured in Hertz (Hz). Common values for refresh rates are 30 Hz for televisions, 60 Hz for general purpose consumer-grade computers, and 100 Hz or more for *low latency* displays that are used in specialized applications such as flight simulators and virtual reality headsets. Most humans start to "see" motion from a sequence of still images at around 24 Hz, with the motion getting smoother and smoother as the refresh rate goes up until around 120 Hz, at which point our ability to reliably pick the higher of two refresh rates disappears (although some rare humans retain this ability up to 150 Hz).

WINDOWS In most operating systems, programs that run with standard privileges do not have full access to the framebuffer. Instead, a program with graphical output requests the operating system to reserve a portion of the display, usually rectangular in shape, for the purposes of display. These are usually (but not always!) known as *windows*. Each window has its own framebuffer, distinct from the global framebuffer for the whole display, and the operating system in cooperation with the display hardware keeps each window's framebuffer in sync with the display's framebuffer. In most systems windows may not be fully visible: they may be partially occluded by other windows, hidden entirely in an iconified or "minimized" form, or, in some fancier cases, blended together with the contents of other windows by the use of transparency, shadows, or other effects. It is the job of the *windowing system* (usually part of

[1] Note that this is different from the response delay *of* humans, which is the amount of time it takes a human to sense, process, and respond to stimuli. In most humans, this is approximately 200 ms.

[2] With only $2^8 = 256$ different intensities per color channel, it is very difficult to depict sharp contrasts as would be created with a bright light (such as the sun) in the background, so we are increasingly transitioning to *high dynamic range* (HDR) displays that have 10 or more bits per color channel, making 32 bits no longer sufficient to store any given pixel.

the operating system) to determine what parts of a window are actually visible, and how to composite the contents of that visible part with the rest of the windows to assemble the contents of the display you actually see.

Double buffering To avoid glitches in the output, it is essential that the framebuffer is not modified as it is being refreshed by the display hardware. Concurrent modifications of the framebuffer would cause part of one frame to be combined with part of another frame. In particular, since most displays are refreshed row-by-row starting from the top, a concurrent modification would have the top portion of one frame mixed with the bottom portion of a different frame, with a clearly discernible line showing the discontinuity. This is known as *screen tearing* and is a highly irritating bug.

Thus, most graphics libraries and windowing systems prevent the framebuffer being displayed from being modified at all. In its most common form, this is achieved with *double buffering*, i.e., with two framebuffers. The *front buffer* is the buffer that is shown on the display during a refresh. This buffer is immutable: its contents are not allowed to be changed by programs. Instead, a mutable second buffer, the *back buffer*, is used to *render* the contents of the next frame. This buffer can be freely modified by programs; it can even be cleared and recreated any number of times. When the entire frame has been assembled in the back buffer, the two buffers are *flipped*: the back buffer is made immutable and is re-designated as the front buffer, while the old front buffer is made mutable and becomes the new back buffer.

Of course a flip must not happen in the middle of a refresh, for that would reintroduce the problem of screen tearing that double buffering was designed to prevent in the first place. In fact, it makes no sense to flip the buffers at any time except right after a refresh. The program must assemble the next frame in the time between one refresh and the next. If the next frame is computed too late, then the program will have to sit out one refresh cycle as the display repeats a frame; this would make the display appear sluggish. Dually, if the next frame is computed before the next refresh begins, then the program will have to wait until the display catches up. If it does the flip before the refresh, one or more frames will be *dropped* and the display will appear jumpy and disjointed. The rate at which the framebuffers are flipped is called the *frame rate*. For optimal results, the frame rate should match the refresh rate: this is sometimes referred to as "*VSync*" to signify that the frame rate is synchronized to the *vertical refresh rate*.[3]

Tracking time There are two ways to synchronize the flips of the framebuffer with the refresh rates of the display. The first is to have the display driver perform the flip itself. This has many problems: chiefly, the program drawing to the display doesn't necessarily control the entire display (unless it is running "fullscreen"). Having the display driver itself go hunting among programs to flip their individual framebuffers is problematic as well: it not only slows the driver down, but requires very precise and accurate book-keeping. Moreover, not all windows may be using double buffering.

This is why it is much more common to synchronize the rates using the second method: a *clock*. All modern processors come equipped with a global clock that is accurate to a few nanoseconds, and this clock is moreover readable by all programs. This clock is used to make sure that the framebuffers are flipped no more often than once every $1000/R$ ms, where R is the refresh rate: for 60 Hz, this means a refresh approximately every 17 ms. To sustain this frame rate, the program must generate each frame within 17 ms, which puts a cap on its complexity: the CPU and the graphics hardware (the *GPU*) must, working together, meet this deadline.

[3] The term *vertical refresh rate* dates from the era of cathode ray tubes, and for all practical purposes the "vertical" is irrelevant today and can be ignored.

6.2 EVENT LOOPS

In parallel with creating the frames for the display, an interactive program may also handle user input, communicate over a network, play sounds, perform computations, and so on. Since everything has to fit within the time to compute a single frame, it is important to carefully budget each aspect of the computation. The standard programming technique for this is an *event loop*, sometimes also called the *main loop*, that has the following general structure, written in pseudo-code:

```
while stuff_to_do():
    process_events()
    update_internal_state()
    redraw_frame()
    wait_for_refresh()
    flip_buffers()
```

Let us tackle each item in turn, illustrating it using the `pygame` library, which we start by importing.

```
1 import pygame as pg
```

The loop condition The program runs until it is instructed to terminate, either by the user taking some kind of "exit program" action or because the window is closed by some external means such as by clicking on the close window button in the window's title bar. In other words, the loop condition just checks the value of a Boolean variable that records whether the program is ready to finish or not.

```
1 done = False
2
3 while not done:
4     # Rest of the event loop goes here
5     #
6     # Eventually something sets done to True
```

Processing events An interactive program may respond to input from a wide variety of sources: the keyboard, mice, touch-screens, speech-to-text systems, vibration sensors, GPS sensors, etc. Each of these inputs is handled *asynchronously* by the Python runtime: as soon as the program receives the event, an event object is added to a processing queue.[4] If these events are not eventually processed, they will linger in the queue forever. Such events are not processed immediately as they arrive because a given event has only a small window in which to be handled, during which other input events are normally blocked. Thus, the code that adds events to the queue is heavily optimized to lower the risk that events are missed.

Therefore, the first task in the event loop is to *process* the events by removing the event objects from the queue and performing any relevant computations. In `pygame` the `event.get()` function removes the next oldest event from the queue and returns it as an object that contains a `.type` data attribute (which is an `int`) defining the kind of event it is, and other possible data attributes depending on its type. Here, for instance, is how we would detect the key press event (`ev.type == pg.KEYDOWN`), and use the `.key` attribute to determine which key was pressed. If the Escape key is pressed, the `done` flag is set to `True` to exit the event loop.

[4] If you've encountered this concept before, it is indeed done with *interrupts*, but it's beyond the scope of this course.

```
done = False
while not done:
    # Process all the events
    for ev in pg.event.get():
        if ev.type == pg.KEYDOWN and ev.key == pg.K_ESCAPE:
             done = True

    # Rest of the event loop
```

Any event of a different type will simply be discarded by the conditional statement on line 5.

UPDATING INTERNAL STATE Once there are no further events to process, the next step is to make all time-dependent updates to the objects in the program. This means things like advancing the frames of an animation or video, or things like changing the positions, velocities, etc. of things that are in motion. If the program has an underlying physical simulation, then that simulation is advanced in this step, which can handle such things as collisions, attraction (gravitational, magnetic, etc.), springs, forces, etc.

To give a simple example, imagine that we have a collection of rectangles that are moving with random velocities in the 2-dimensional plane, which are represented as instances of the following `Box` class:

```
class Box:
    def __init__(self, x, y, vx, vy):
        # (x, y) is the coordinate of the top left corner
        self.x = x
        self.y = y
        # the velocity of the box in units of pixels/millisecond
        self.vx = vx
        self.vy = vy

    def update(self, millis):
        # millis is the number of milliseconds since the last call to .update()
        self.x += int(self.vx * millis / 5)
        self.y += int(self.vy * millis / 5)
        # we divide by 5 to slow it down for observation
```

Imagine that we start with a collection of 5 boxes with random positions and velocities:

```
boxes = [Box(random.randint(0, 100), random.randint(0, 100),
             random.uniform(-2, 2), random.uniform(-2, 2)) for _ in range(5)]
```

In our event loop, we would call the `.update()` member function of a `Box` to update its `x` and `y` coordinates. This function takes an additional parameter which is the number of milliseconds that has elapsed since the previous call to the function. This parameter is explicitly tracked in the updated event loop. To keep track of time, we use the `datetime()` function from the `datetime` module which will always return an object that represents the current time, which is accurate to microseconds. (There are more accurate clocks, but microseconds suffices for our purposes.) In every run of the loop, we recompute the current time and then the time difference from the previous iteration of the loop; we then convert this time difference to microseconds and divide by 1000. (The `.microseconds` data attribute is implemented using *properties* that we will cover later in this course – REFERENCE TO BE ADDED.)

```
from datetime import datetime

t = datetime.now() # current time
while not done:
    # event handling code from before

    cur_t = datetime.now()
    delta_t = cur_t - t
    for box in boxes:
        box.update(delta_t.microseconds // 1000)

    t = cur_t

    # rest of the event loop
```

Generally speaking the update to the state of the simulation will be a lot more complex—in fact, the most complex part of the simulation—and will require careful planning to handle situations where there is feedback. For instance, if we have two boxes connected by a spring, then the motion of each box will change the forces that act on the other box, affecting the latter's motion, which in turn will affect the former's motion and so on.

RENDERING A FRAME So far we have not specified how a box is drawn, just its position and velocity. Once the positions of the boxes have been updated, we need to draw them onto the *window*. The window itself is created with the `display.set_mode()` function that takes a (width, height) as argument. At the point of rending a frame, we can obtain the current width and height of the window using `.get_width()` and `.get_height()` functions.

For now, let's draw the boxes as 20 pixel wide squares, and interpret its coordinates as being given modulo the width and height of the window, which has the effect of the boxes appearing to "wrap around" in the window (exiting from the right causes it to enter from the left, etc.). This draw function can be added to the `Box` class.

```
yellow = pg.Color('yellow')

class Box:
    ### include everything from before here

    def render(self, window):
        w = window.get_width()
        h = window.get_height()
        # make the coordinates positive
        while self.x < w: self.x += w
        while self.y < h: self.y += h
        # draw it as a 10x10 yellow square
        pg.draw.rect(window, yellow, (self.x % w, self.y % h, 20, 20))
```

Careful readers will note that we are being a bit sloppy in our wraparound behavior, since if a box is partially outside the right or bottom edge of the window, the obscured portion doesn't appear on the other side of the window.

In order to write the rendering portion of the loop, we have to create the window and set it to be double-buffered. Note that in order to create this window, we need to initialize the `pygame` library using the `pg.init()` function; dually, after the end of the event loop, we run `pg.quit()` to remove all windows

and release all resources used to draw. The `.quit()` call is not strictly needed if the Python program were to actually exit after the event loop, but it is still good hygiene to leave it in your code, particularly if you want to be able to test it in a running instance of Python.

```
# initialize pygame
pg.init()

# 800x600 pixel double-buffered window
window = pg.display.set_mode((800, 600), pg.DOUBLEBUF)

### boxes and timing

done = False
while not done:
    ### event handling and updates as before

    window.fill(pg.Color('black')) # clear the window
    for box in boxes:
        box.render(window)

# quit pygame
pg.quit()
```

Flipping buffers If you run the above code, you will only see a blank window, since all the updates are being made to the back buffer. To flip the buffers we have to use `pg.display.flip()`, which we do at the end of every iteration of the event loop.

```
while not done:
    ### event handling, updates, and rendering

    pg.display.flip()
```

Syncing with the refresh rate The above event loop runs as fast as possible, which causes lots of calls to `.flip()` that will not actually result in an updated display. The display hardware never refreshes the screen more often than its fixed refresh rates, so all these extra frames are just wasted processor cycles. Achieving a "frame rate" higher than the refresh rate of the display is hardly ever useful.

To synchronize with the display rate we can reuse the `datetime()` feature as we did earlier, but `pygame` has a built in convenience function to choose any arbitrary frame rate. This is the `pg.time.Clock` class whose instances have one important member function: `.tick()`. This function takes the desired frame rate R (in Hz) as input and its purpose is to go to sleep until the remainder of the current $1/R$ time slice has expired. At that time the function returns the number of milliseconds that it slept. The full code listing with this utility class is shown in Figure 6.1.

6.3 GOING FURTHER

While `pygame` is a simple library for quick prototyping, it is not the kind of library that is usable for serious interactive graphics. Nowadays the Pyglet library is recommended, although it has a higher level interface that obscures the event loop. Pyglet also depends on object-oriented programming, which is covered in Chapter 1.5.

```
import pygame as pg

black = pg.Color('black')
yellow = pg.Color('yellow')

class Box:
    def __init__(self, x, y, vx, vy):
        # (x, y) is the coordinate of the top left corner
        self.x = x
        self.y = y
        # the velocity of the box in units of pixels/millisecond
        self.vx = vx
        self.vy = vy

    def update(self, millis):
        # millis is the number of milliseconds since the last call to .update()
        self.x += int(self.vx * millis / 5)
        self.y += int(self.vy * millis / 5)

    def render(self, window):
        w = window.get_width()
        h = window.get_height()
        # make the coordinates positive
        while self.x < w: self.x += w
        while self.y < h: self.y += h
        # draw it as a 10x10 yellow square
        pg.draw.rect(window, yellow, (self.x % w, self.y % h, 20, 20))

boxes = [Box(random.randint(0, 100), random.randint(0, 100),
             random.uniform(-2, 2), random.uniform(-2, 2)) for _ in range(5)]

# initialize pygame
pg.init()

# resizable 800x600 pixel double-buffered window
window = pg.display.set_mode((800, 600), pg.DOUBLEBUF)
clock = pg.time.Clock()

done = False
millis = 0
while not done:
    # Process all the events
    for ev in pg.event.get():
        if ev.type == pg.KEYDOWN and ev.key == pg.K_ESCAPE:
            done = True
    # Clear
    window.fill(black)
    # Update
    for box in boxes: box.update(millis)
    # Render
    for box in boxes: box.render(window)
    # Synchronize
    millis = clock.tick(60) # 60 is the refresh rate
    # Flip the buffers
    pg.display.flip()

pg.quit()
```

Figure 6.1: Flying boxes with random velocities

7 COMPACT REPRESENTATION

In this chapter we will cover a few techniques for representing data and data structures in a compact form using the building blocks we have already seen: lists, dictionaries, classes, and so on. We will then investigate the mathematical foundations of data: what does it mean to encode data, how efficient is an encoding, and how to quantify such things as the "*encodability*" of data.

7.1 LINEAR REPRESENTATIONS

In Chapters 4 and 5 we have seen a few techniques for representing trees and graphs using recursive representations built from nodes and successor (and sometimes predecessor) arcs. While these representations are easy to build and work with, they are not the most compact in terms of memory usage. Modern computers lay out their memory in such a fashion that it is most beneficial to have closely related data also reside next to each other in memory, so that the entire group of related objects can be loaded at once into the high speed memory *caches* located right next to the register banks and arithmetic-logic units of a processor as part of the same chip. Thus, in some cases it is better to *linearize* the data into lists/arrays. We will see two examples in this section.

7.1.1 *Disjoint sets: the union-find structure*

In Chapter 2 we saw how to represent sets of numbers as *bitsets*, using the bits of a single `int` to determine if a given element was in or out of the set. However, this representation was wasteful for sparse sets; for example, the singleton $\{1000\}$ would be represented with a 1001-bit word with only one of its bits set to `1`. It would have been more efficient to represent it as the list `[1000]`, which uses only 72 bytes, as opposed to the bitset representing $\{1000\}$, which uses 160 bytes.

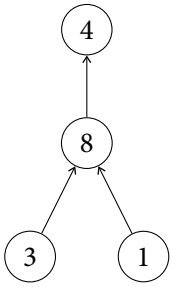
```
>>> import sys
>>> sys.getsizeof([1000])
72
>>> sys.getsizeof(1 << 1000)
160
```

Consider the problem of representing a collection of *disjoint sets* of numbers. Every number in the collection is a member of exactly one of its constituent sets. Once again we could use a list of bitsets, one bitset per disjoint set, but it is difficult (i.e., computationally expensive) to enforce the invariant that each element is in exactly one of the sets. We could do slightly better by uniquely numbering the constituent sets and having each element of the set record its set number: this would make it easy to determine when two numbers belong to the same set or not: just check their associated set numbers.

Now imagine that we want to be able to join two of these disjoint sets into the same set, i.e., to perform a union. To do this in constant time, we cannot give the joined set a new number and then update the association from the members of the set to this new number, for that would take time linear in the cardinality of the new set.

UNION AND FIND The *union-find* data structure provides a clean solution: instead of having the elements of the set point to some numbering of the set, just have them point to some other element of the

set as the parent in a *tree* structure. For instance, to represent the set $\{1, 3, 4, 8\}$, we might imagine the following pointers:



The root of this tree does not have a parent; indeed, a node does not have a parent *if and only if* it is a root. We can therefore take the root of the tree as the designator for this entire set. Note that it is important that the parent relationships actually determine a tree, for otherwise we may have multiple designators for a set (for a DAG), or even no designator at all (if the graph has cycles).

To represent such a set compactly, we can use a dictionary, where the keys of the dictionary are the elements of the set, and the values are the parents for each element. (Remember that dictionaries are actually implemented as hash tables, which is a compact list/array-like structure.) The above tree, for instance, would be the dictionary `{1: 8, 3: 8, 8: 4, 4: None}`. Furthermore, multiple such trees (called a *forest of trees*) can exist in the same dictionary, and the number of sets in the collection will be exactly the same as the number of roots, i.e., the number of keys that are mapped to `None`.

Given such a collection `S` of disjoint sets, we can find the designator for any element `a` by repeatedly traversing its parent arcs until we reach the root. Here it is, written in a recursive style.

```
def find_root(S, a):
    """Return the designator of the element a in the disjoint
    set collection S"""
    if S[a] is None: return a
    return find_root(S, S[a])
```

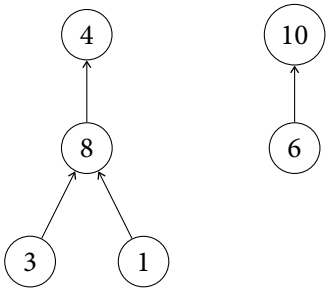
To see if two elements `a` and `b` are in the same set in the collection of disjoint sets, we merely have to see if they have the same designators.

```
def is_same_set(S, a, b):
    """Are a and b in the same set in the disjoint set collection S?"""
    return find_root(S, a) == find_root(S, b)
```

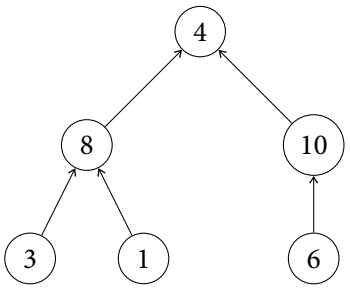
The union operation can be defined as taking two elements `a` and `b`, and replacing the sets they belong to with their union. This is the same as finding the designators of `a` and `b`, and making one of them the parent of the other (assuming they are different).

```
def union(S, a, b):
    """Perform a union if needed so a and b end up in the same
    set in the disjoint set collection S"""
    ra = find_root(S, a)
    rb = find_root(S, b)
    if ra != rb: S[ra] = rb
```

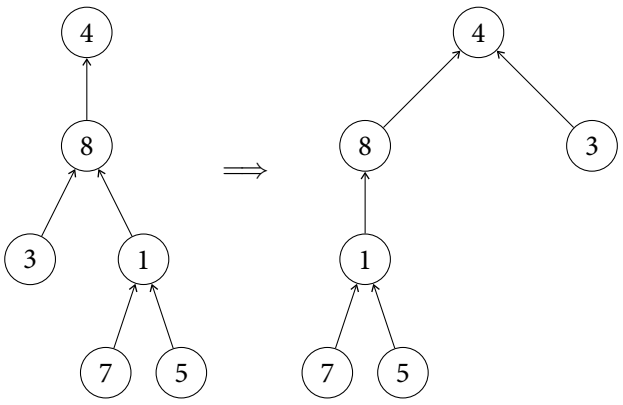
As an illustration, imagine that we want to join 3 and 6 in the two trees below.



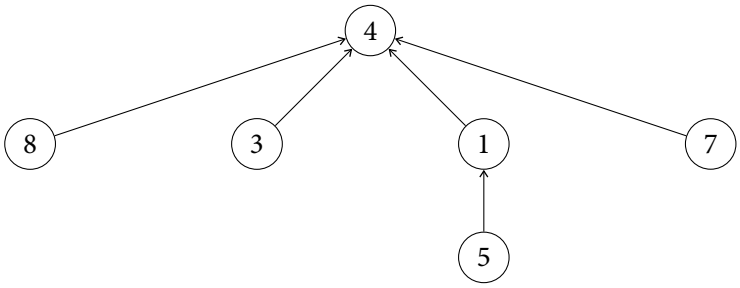
The `union()` function would find the two designators 4 and 10, and then set one to the parent of the other; for instance:



PATH COMPRESSION When we find the designator of a given element, we may follow a chain of parent links to get to the root of its tree. The next time we need to find its designator, we need not follow the same path as before: we can modify its parent edge so we directly jump to the root instead. For instance, from the following tree if we find the designator for 3, we can transform the tree by making 3 a direct descendant of the root without altering the designators of any of the nodes of the tree.



Next, if we were to find the root of 7 we can similarly make it a direct descendant of 4; but, not only that, we can make 1 and 8 the direct descendants of the root as well, since to find the root of 7 we had to (recursively) find the root of 1 and 8 as well! So, we get the following much shallower tree:



This optimization of `find_root()` is known as *path compression*, and is implemented as follows:

```
def find_root(S, a):
    """Return the designator of the element a in the disjoint
    set collection S, compressing paths as we go"""
    if S[a] is None: return a
    root = find_root(S, S[a])
    S[a] = root
    return root
```

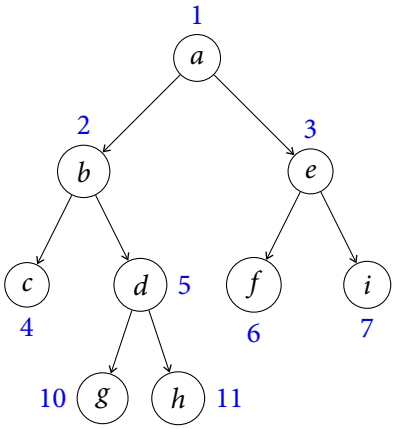
None of the other functions—`union()` and `is_same_set()`—needs to be modified.

RANKING The `union()` operation makes one tree a direct descendant of the root of the other tree. Using the intuition that it is better for the trees to be as shallow as possible, we can use the heuristic that the deeper tree should make the shallower tree its descendant. However, the data structure does not contain any depth information, and to compute the depths would require traversing the entire collection of elements across all the different disjoint sets. So, we need to maintain this depth information on an element by element basis as we go. This optimization is known as *ranking*, and together with path compression makes this union-find data structure exceptionally performant, making it take effectively constant time for collections of any reasonable size. Indeed, the asymptotic growth of the running time of this function is related to the inverse of the Ackermann function. The Ackermann function grows faster than any function expressible using elementary arithmetic operators, so its inverse grows slower than any function expressible using iterated logarithms.

7.1.2 *Linearized binary trees and binary heaps*

Let us consider again the question of representing *binary* trees. You have a already seen a few implementations of such trees, such as at the end of CSE 101 and during Chapter 4. In this section we are going to show how to represent such trees using *lists*.

THE ISOMORPHIC LIST To motivate the list representation, consider the following tree, where the data in the nodes of the tree is arbitrary (we've used letters $a, b, \ldots$).



Let us number the root node with the index 1, shown in blue. Then, we will follow the following numbering scheme: for a node numbered k, the number of the root of its left subtree (if it exists) is $2k$, while that of the root of its right subtree is $2k + 1$. This numbering is then recursively carried out for the

two descendant subtrees. We obtain the numbering shown above in blue. We immediately observe a few properties (that you should quickly prove for yourself to verify your intuitions):

1. Every node is given a unique number.
2. Not all numbers starting from 1 are necessarily used. Here, for instance, there is no node with numbers 8 or 9.
3. For all nodes except the root node, if it is numbered n then its parent is numbered $\lfloor n/2 \rfloor$, i.e., n `>> 1` in Python.
4. The nodes on level $k+1$ of the tree all have numbers that are strictly larger than the nodes on level k of the tree.

The list representation of this such a tree is given in terms of the node numbering. We create a list where the ith index contains the node that corresponds to the number i. Any index below the largest numbered node in the tree that doesn't have a corresponding node in the tree is replaced with `None`. So, the list to represent the above tree is:

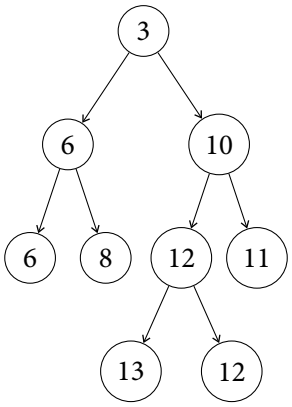
```
[None, a, b, e, c, d, f, i, None, None, g, h]
```

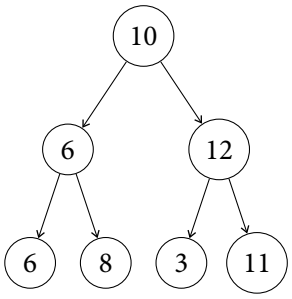
From such a list we can easily reconstruct the tree using fact 3 above. Hence, this is an *isomorphic* representation of the tree. Moreover, by fact 4, it is the case that any element that is added to the end of the list corresponds to adding it as a new leaf of the tree. If none of the entries in the list except for the very first one (for index 0) is `None`, then the tree has full levels for all except the last level, which is being filled in from the leftmost child to the rightmost child.

Binary Heaps A *heap* is a structure that stores a collection of mappings from keys that have a total ordering to values, supporting the following main operations:

- *Size*: how many mappings are there in the heap?
- *Find-Min*: find the smallest key that has a mapping in the heap.
- *Delete-Min*: remove a mapping for the smallest key in the heap.
- *Add*: insert a new (key, value) mapping to the heap.

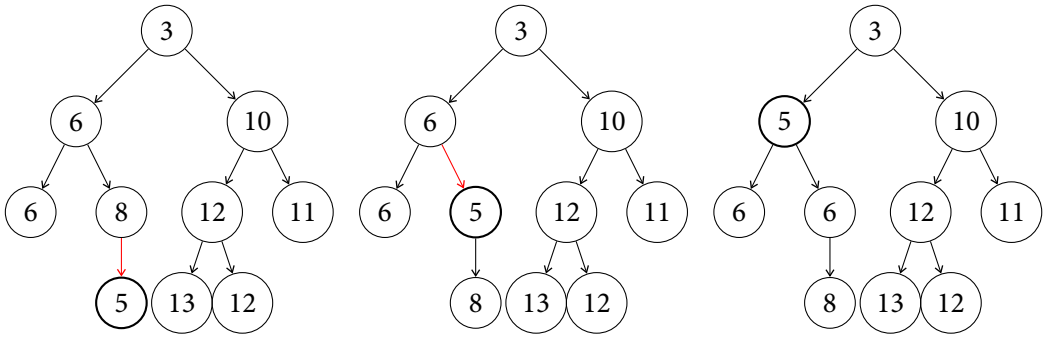
A particular example of a heap structure, called *binary heaps*, can be done very simply by exploiting the isomorphic list structure. A binary heap is a binary tree where the nodes store the (key, value) pairs, and two nodes are compared based on their keys. The tree satisfies the following *min-heap* invariant: in any subtree, the key of the root node of the subtree is the smallest key stored in any node in that subtree. Here are two trees where the left tree satisfies the min-heap invariant while the right does not (we have shown only the keys in the nodes).



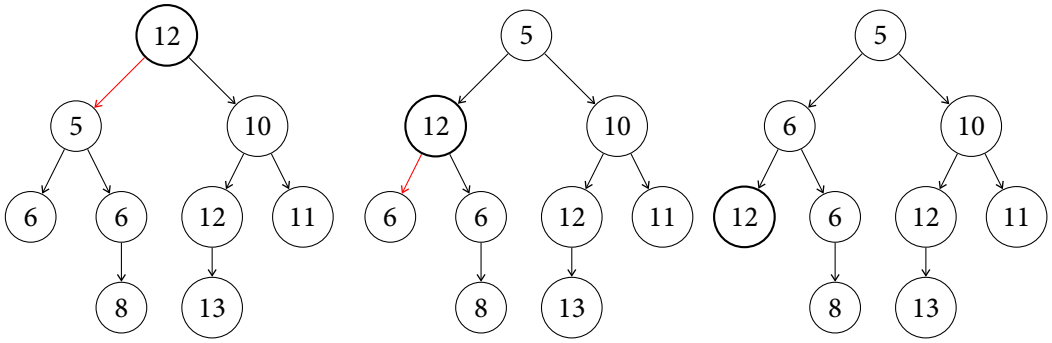


Here is how the heap operations are implemented on top of binary trees with the min-heap invariant.

- *Size*: this is simple: just do any traversal of the tree, such as a depth-first traversal, and count the number of visited nodes. The complexity is $O(n)$ where n is the number of nodes in the tree, but this can be improved to $O(1)$ if each node stores some additional data about the number of nodes in the subtree rooted there.
- *Find-Min*: this is trivial: the root of the tree already contains the smallest key in the entire tree.
- *Add*: this is not entirely trivial, because the key we add can only be added as a new leaf of the tree, which may break the min invariant temporarily. Fortunately, there is a simple *percolate up* operation that we can perform to restore the invariant: repeatedly compare the added node to its parent, and swap the node with its parent if the parent does not have a smaller key. Here is an illustration of adding the node 5 to the heap depicted above, drawn with a thicker border, and the necessary percolations shown with red arcs.



- *Delete-Min*: as you might expect, this is implemented by removing the root of the heap – but what do we replace it with? One simple technique is to detach one of the leaves of the tree and make it take the place of the newly removed root, but this again would make it potentially violate the min-heap property. To restore it, we *percolate down* the root node by repeatedly comparing it with the roots of its immediate subtrees, and swapping it with the smaller of the two if it is also smaller than the root. To illustrate, here is what happens when we perform this operation on the above heap, removing the 3, after which we promoted the node 12 (any other leaf would have also worked):



IMPLEMENTING BINARY HEAPS WITH LISTS Let us now see how we may implement the above operations defined on trees in terms of the isomorphic lists.

- *Empty Heap*: the heap with no nodes corresponds to the singleton list `[None]`.

```
def empty_heap():
    return [None]
```

- *Size*: we will make sure that there is only a single instance of `None` in the list, in index 0. This will mean that the number of nodes in the heap will be 1 less than the length of its isomorphic list.

```
def size_heap(H):
    return len(H) - 1
```

- *Min-Heap*: the root of the tree always has index 1 in the isomorphic list, so:

```
def min_heap(H):
    if len(H) > 1:
        return H[1]
```

- *Add*: to place make a new leaf is the same as appending a new element to the isomorphic list. Then, repeatedly compare it to the key at the index of its parent.

```
def add_heap(H, k, v):
    ix = len(H)                # index of entry to add
    H.append((k, v))           # append to the list
    while ix > 1:              # stop and index 1, which is the root
        pix = ix // 2          # index of parent
        if H[ix][0] >= H[pix][0]: break
        # otherwise swap and keep going with parent
        H[ix], H[pix] = H[pix], H[ix]
        ix = pix
```

- *Delete-Min*: Now we just take the last element of the list and put it at the front, and then percolate it down by comparing it to the keys of its two children nodes.

```
def del_min_heap(H):
    if len(H) == 1: raise ValueError   # empty heap
    H[1], H[-1] = H[-1], H[1]  # swap the first and last elements
    ret = H.pop()              # this will be returned
    ix = 1                     # index of node being percolated down
    while ix < len(H):
        swix = ix              # index of node to swap with
        if ix * 2 < len(H) and H[swix][0] > H[ix * 2][0]:
            swix = ix * 2
        if ix * 2 + 1 < len(H) and H[swix][0] > H[ix * 2 + 1][0]:
            swix = ix * 2 + 1
        if swix == ix: break   # no swap, so we're done
        # otherwise swap and keep going with the swapped child
        H[ix], H[swix] = H[swix], H[ix]
        ix = swix
    return ret
```

7.2 COMPACTING LINEAR DATA

We have now seen a couple of ways of reducing tree-like data to a linearized form. In general, any graph can be linearized by just numbering all the nodes from 0 onward, and then making using the numbers as indices of an array (list). This is such a common technique that it is built directly into Python and many other languages, using a process called *marshaling*: to marshal data is to turn it into an array of bytes in such a way that the original object can be restored from it, a process cleverly called *unmarshaling*. We won't cover marshaling further in this course, but you can read about it in the Python documentation.

In the rest of this section we will work with the assumption that Python strings are composed of characters that are representable in bytes, i.e., that a string of length n has exactly n bytes in it. If we wanted to be pedantic, we would have to use `bytes` objects instead of **`str`** objects, but that is a distraction from the main purpose of this section.

Encoding and Decoding Once we have converted data into a sequence of bytes, we may then wonder whether we can compact it further by reversibly transforming it into a different sequence of bytes. To be precise, we are searching for two functions $\mathrm{enc}, \mathrm{dec} : \mathbb{N}_{256}^* \to \mathbb{N}_{256}^*$[1] called the *encoding* and *decoding* functions, such that decoding is the left inverse of encoding, i.e., for all byte sequences $b \in \mathbb{N}_{256}^*$, $\mathrm{dec}(\mathrm{enc}(b)) = b$.

An encoding function is called a *compression function* if for all $b \in \mathbb{N}_{256}^*$, $|\mathrm{enc}(b)| \leq |b|$ where $|b|$ is the length of b. Note that it is not possible to make the size strictly smaller always, i.e., $|\mathrm{enc}(b)| < b$ cannot be the case for every b. To see why, assume that this is the case. Then, we can compress any data b of any size down to a single byte by using the strictly decreasing chain:

$$|b| > |\mathrm{enc}(b)| > |\mathrm{enc}(\mathrm{enc}(b))| > \cdots .$$

Note that there are more than 256 different bit sequences: even with two bytes, we can distinguish $2^{16} = 65536$ different bit sequences. Therefore, by the pigeonhole principle, there must be two different byte sequences that get compressed down to the same byte by the chain above. Since an encoding function has a left-inverse, we must be able to apply a sequence of $\mathrm{dec}()$s to this byte to be able to get both these different input byte sequences, which would contradict the fact that dec is a function.

Strictly speaking, our definition of compression function above is more commonly known as *lossless compression*, which suggests that there is such a thing as *lossy compression*. This topic is beyond the scope of this course. Briefly, however, lossy compression has the property that decompressing does not yield exactly the original byte sequence, but a byte sequence that is "close enough" based on a quality metric. Such compression techniques are used for such kinds of data as bitmap images and sounds, because we can use human cognition to drive the "close enough" metric: if two sounds or two images cannot be distinguished by most humans, then it doesn't matter if one of them is mapped to the other after an encoding/decoding round-trip.

Run-Length Encoding (RLE) One of the simplest encoding functions is *run-length encoding* that looks for sequences of repetitions (a "run"), and replaces an entire block of repetitions with one occurrence followed by a count of the number of repetitions (the "run-length"). As a simple example, the string `'aaabbcccc'` would be encoded as `'a3b2c4'`, which reduces the size by 3 bytes. An alternative is to encode it as `'a\x03b\x02c\x04'`, in which every alternate byte is a literal number encoded as a byte, which

[1] $\mathbb{N}_{256}^*$ is the set of *sequences* of natural numbers modulo 256 (i.e., from 0 to 255). 255 is the largest natural number representable in a single byte.

is written with the `\x` escape sequence in Python.[2] Whatever the encoding, we must carefully account for the possibility that the bytes used in the output may also occur in the input. Imagine, for instance, trying to compute the RLE of `'a3b2c4'` itself; if we naively write its encoding as `'a131b121c141'` then it may be interpreted as a sequence of 131 `a`s, then 121 `b`s, and finally 141 `c`s.

Note that RLE is not guaranteed to reduce the size of the input. In fact, the more random the input is, the more likely it is that RLE will actually *expand* the size of the string. Take, for instance, the string `'mxyzptlk'`; its encoded string is `'m1x1y1yz1p1t1l1k'`, which is twice the length. To prevent this, it is common to start counting the run from the *second* character onward, leaving the first character untouched. Thus, for example, we could encode `'baaababbbbba'` as `'baa2babb4a'` where the encoding kicks in only after we see a sequence of at least two identical bytes. This will prevent a size explosion for very random input strings while retain the benefits of RLE for more uniform strings. However, note that even this encoding—and any encoding that only has a fixed amount of "lookahead"—can be defeated by just having runs of only that fixed length. For instance, the above technique is unable to handle `'mississippi'`, which it would blow up to `'miss1iss1ipp1i'`, an *increase* of 4 bytes.

Another issue with RLE is that it only looks at repetitions of individual characters, when in fact repetitions of *sequences* of characters is more common. For instance, the previous example, `'mississippi'` has a repetition of the substring `'iss'`, so if we were to denote runs of words with parentheses we could have encoded it as `'m(iss)2ippi'`. (Of course we would then need to figure out what happens if there are literal `'('` or `')'` bytes in the *input* string.)

Huffman Codes Suppose we want to encode a string where each byte of the string comes from a fixed set, say $\{$`'0'`, `'1'`, ..., `'9'`$\}$. How many bytes do we need to encode a string n bytes long? To represent 10 different things, we could make do with 4 bits, which can represent the range $0, 1, \ldots, 15$. Therefore, using half a byte per encoded byte, we could encode the entire input in $n/2$ bytes. Is that the least number of bytes?

More concretely, what is the least number of bytes needed to encode a given input string? The answer clearly depends on the input string that we are trying to encode. Imagine if the input string were just a sequence of `'0'`s: `'000...0'`. We could encode this as just the length of the string, since every byte of the input string is completely predictable, i.e., `'0'`. How about an input string that contains only `'0'`s and `'1'`s? In this case, we could encode `'0'` using the *bit* `0`, `'1'` as the bit `1`, and thereby pack 8 input bytes into a single encoded byte, making the encoding 8 times smaller than the input. Now how about if the input contains `'0'`s, `'1'`s, and `'2'`s, with `'0'` being the most frequent byte? In this case, if we want to have the most compact encoding, we should pick the following bit patterns:

```
'0':0
'1':10
'2':11
```

With this choice, here is how we can write our decoder: as we are reading a stream of bits, whenever we read a `0` bit, we can output a `'0'`. If we read a `1` bit, we look at the next bit and if it is a `0` bit we output `'1'`, otherwise we output a `'2'`. This is guaranteed to be the most compact encoding possible, since we cannot distinguish between three things with only one bit (by the pigeonhole principle), and we moreover avoid "wasting" the bit sequence `00`.

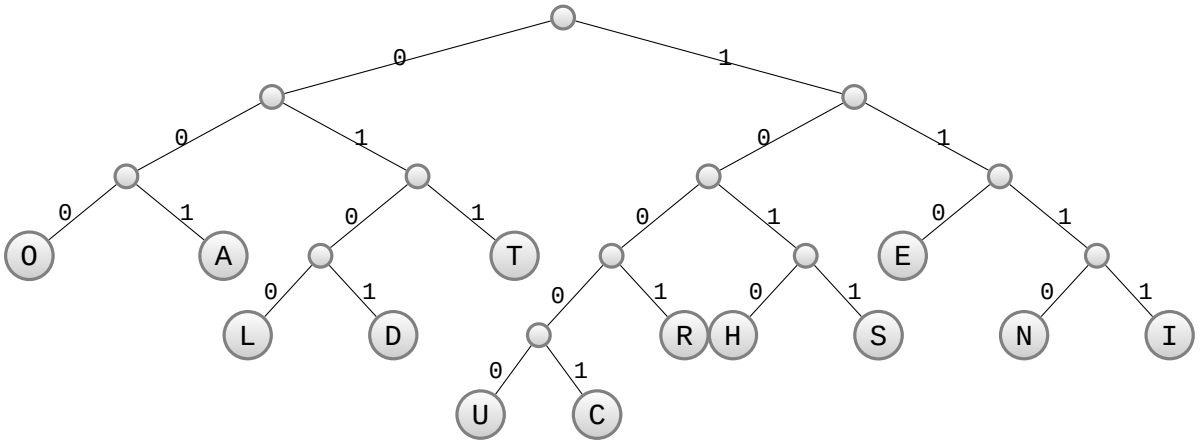
This kind of encoding of bytes in terms of bit sequences that guarantees the most compact encoding possible is known as a *Huffman code*, named after the computer scientist David Huffman who laid the

[2] The format of an `\x` sequence is the literal characters `\x` followed by a two character Hexadecimal sequence, so the range of representable numbers is `\x01`, `\x02`, ..., `\xfe`, `\xff`.

foundation for the field of *information theory*. Formally, a Huffman code is a mapping from a given finite set of inputs (called the *alphabet*, but think bytes), where each input has an associated *weight* (think: frequency), to bit sequences that has the following three properties:

- *Injection*: different inputs must be mapped to different nonempty bit sequences.
- *No common prefix*: the bit sequences associated to two different inputs must be such that neither sequence is a strict prefix of the other.
- *Compactness*: if an input x has a higher weight than an input y, then the bit sequence associated to x must be *no larger than* that associated to y.

It is common to think of a Huffman code as a binary tree where an edge from a parent to its left child is labeled `0` and the edge to its right child is labeled `1`. Each leaf of the tree corresponds to a unique element of the set of inputs. The bit sequence associated to one such input is the sequence of labels on the path from the root of the tree to that leaf, read off as a a bit sequence. As an example, here is a Huffman code for the set of input bytes {`'A'`, `'C'`, `'D'`, `'E'`, `'H'`, `'I'`, `'L'`, `'N'`, `'O'`, `'R'`, `'S'`, `'T'`, `'U'`}, which are the 12 most common letters in the English language. Moreover, these letters in the order of most to least frequent are: `ETAOINSHRDLCU`, so the weight we assign to each of these input bytes is its position in this frequency list, reading from right to left. In other words, `'E'` has the highest weight and `'U'` has the lowest weight. The Huffman code tree is as follows.



It is easy to see that the common prefix condition is satisfied, since only the leaf nodes are associated to input bytes. The injection property also holds because there are no dangling leaves and each input byte occurs in exactly one leaf node. The compactness property is a little bit harder to see: level 2 of the tree contains the leaf nodes `OATE` which is a permutation of the first four letters in `ETAOINSHRDLCU`; then level 3 contains `LDRHSNI` which is a permutation of the next 7 letters; and finally level 4 contains `UC` which is a permutation of the last two letters.

To encode a word such as `'CHEATSHADES'`, we would go letter by letter and replace each with the bit string from the root to the corresponding letter in the Huffman code tree. For instance, `'C'` is accessed by going right-left-left-left-right, which is the bit sequence `10001`; next `'H'` is `1010`; and so on, giving the final encoded sequence:

```
10001 1010 110 001 011 1011 1010 001 0101 110 1011.
```

We can regroup these into bytes to get:

$$\underbrace{\texttt{10001101}}_{\texttt{0x8d}}\ \underbrace{\texttt{01100010}}_{\texttt{0x62}}\ \underbrace{\texttt{11101110}}_{\texttt{0xee}}\ \underbrace{\texttt{10001010}}_{\texttt{0x8a}}\ \underbrace{\texttt{11101011}}_{\texttt{0xeb}}.$$

```
def create_huffman_tree(bs):
    """Return the Huffman tree for the bytes bs"""
    probs = get_probabilities(bs)
    H = empty_heap()
    # create the leaves
    for b in probs:
        add_heap(H, probs[b], b)
    # repeatedly create internal nodes for the least probable
    # internal nodes as long as there are at least two
    while size_heap(H) > 1:
        (lp, l) = del_min_heap(H)
        (rp, r) = del_min_heap(H)
        add_heap(H, lp + rp, {0: l, 1: r})
    # when there is only one tree left, it return it
    return del_min_heap(H)[1]
```

Figure 7.1: Creating a Huffman code tree

Thus, the encoding of the input string `'CHEATSHADES'` is `'\x8d\x62\xee\x8a\xeb'`.

To build a Huffman tree for a given input string, we first compute the probability of encountering each of the bytes of the string, which is just its number of occurrences of each byte in the string divided by the length of the string.

```
def get_probabilities(bs):
    """Get a mapping from each byte in bs to its probability"""
    freqs = {}
    for b in bs:
        if b in freqs: freqs[b] += 1
        else: freqs[b] = 1
    return {b: freqs[b] / len(bs) for b in freqs}
```

Consider each of these bytes to be the leaves of the Huffman code tree. To build the internal nodes of the tree, we first pick the two least probable leaves and then make a parent node joining them, and set the probability of this internal node to be the sum of the probabilities of the two nodes joined. We then repeat this process, joining leaf nodes and subtrees in this fashion until we are left with exactly one tree, which will be the Huffman code tree. Note that we can easily perform the actions of retrieving the two minimal elements of a collection of trees by using the heaps we saw in Section 7.1.2.

In Python, let us use a very lightweight representation of a Huffman code tree where leaf nodes are represented just as individual bytes, and internal nodes are represented a a dictionary mapping the key `0` to the left subtree and the key `1` to the right subtree. For instance, the Huffman code tree shown above would be represented as follows, which is created by the code in Figure 7.1.

```
{0: {0: {0: 'O', 1: 'A'},
     1: {0: {0: 'L', 1: 'D'}, 1: 'T'}},
 1: {0: {0: {0: {0: 'U', 1: 'C'}, 1: 'R'},
         1: {0: 'H', 1: 'S'}},
     1: {0: 'E', 1: {0: 'N', 1: 'I'}}}}
```

7.3 ENTROPY OF DATA

Huffman code trees need to have as many leaves as the size of the alphabet constituting the data. If there are n elements in the alphabet, then the tree will need n leaves, and therefore its minimum depth is $\log_2 n$ (why?). Since the length of the bit sequence for encoding any of the characters is bounded above by the height of the tree, we see that the minimum number of bits needed (in the worst case) to encode a string of length k built from an alphabet of n characters is $O(k \log_2 n)$.

In some cases we can do much better than this theoretical upper bound. For instance, if only one of the characters of the alphabet is actually used, then we can represent it with a very short bit sequence of length exactly 1, and hence the number of bits needed is just $O(k)$.[3] However, if the data is not so predictable, then we end up approaching the worst case estimate. In fact, if the data is uniformly random, meaning that each element of the alphabet is equally likely to appear in the input string, then the Huffman code tree for it will be a balanced tree and all characters will end up using exactly $\log_2 n$ bits in their encoding.

The "compressibility" of data is in some sense related to its randomness or its *entropy*. This is a key concept in the field of *information theory* that arose out of the work of Huffman, but was placed on firm mathematical ground by Claude Shannon. One of Shannon's main contributions was to define a set of postulates about data entropy from which we can derive a formula for calculating the entropy of any given piece of data, from which we can estimate how much further we can (theoretically) compress the data.

Shannon Entropy The derivation of Shannon's formula of entropy from first principles is beyond the scope of this course, but it is closely tied to (and can be seen as a generalization of) Boltzmann's definition of thermodynamic entropy. The key idea is that given an input string d built out of a finite alphabet S, the entropy of d, written $H_S(d)$, is computed as follows:

$$H_S(d) \quad = \quad -\sum_{x \in S} p(x,d) \log_2 p(x,d),$$

where $p(x,d)$ is the probability of encountering the character x in d:

$$p(x,d) \quad = \quad \begin{cases} \dfrac{\text{number of occurrences of } x \text{ in } d}{|d|} & \text{if } |d| > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Usually the alphabet S is understood from context, so we just write it as $H(d)$ instead of $H_S(d)$. Furthermore, in the computation of $H(d)$, if it is the case for some $x \in S$ that $p(x,d) = 0$, then we take the product $p(x,d) \log_2 p(x,d)$ to also stand for 0, even though $\log_2 0$ is not defined. Note that $0 \leq p(x,d) \leq 1$, so $p(x,d) \log_2 p(x,d) \leq 0$; hence, because of the overall $-$ sign, it will be the case that $H(d) \geq 0$.

Entropy calculation examples For the purposes of this example, suppose that S is the set of all lowercase Latin letters, `'a'` – `'z'`.

- *Empty string*: here, $p(x, \texttt{''}) = 0$ for every x, and hence $H(\texttt{''}) = 0$.
- *Predictable string*: assume that $d =$ `'aaaa...a'`, a string containing of repetitions of the character `'a'`. In this case we have $p(\texttt{'a'}, d) = 1$ and for any $x \neq$ `'a'`, $p(x,d) = 0$. So:

$$H(d) = -1 \log_2 1 - \sum_{x \neq \texttt{'a'}} 0 = 0$$

[3] In fact, we can even get away with $O(\log_2 k)$ bits since we don't need record the perfectly predictable letter that appears, just how many times it appears, which is k. A single number k can be represented in binary with $O(\log_2 k)$ bits.

- *Completely random string*: assume that $|d| = n$, and that d contains a uniform distribution of S, i.e., each letter in S occurs exactly $n/26$ times in d. Hence, $p(x, d) = \frac{n/26}{n} = \frac{1}{26}$. Plugging it into the formula for entropy, we get:

$$H(d) = -\sum_{x \in S} p(x, d) \log_2 p(x, d) = -\sum_{x \in S} \frac{1}{26} \log_2 \left(\frac{1}{26}\right) = -\log_2 \left(\frac{1}{26}\right) = \log_2 26$$

AN INTUITIVE MEANING OF ENTROPY What does a value such as an entropy of 0 or $\log_2 26$ actually mean? One intuitive way to think of it is that this is the amount of *information* contained in one unit—each element of the alphabet—of the data. If we were to represent it using bit strings, the number of bits we would need to encode d in the most compact way possible is $|d|\ H(d)$. Remember that entropy measures the most compact representation possible using any algorithm, known or unknown, which means that it is not possible to compress data past this entropy bound.

If $H(d) = 0$, which means that d is completely predictable, then the most compact representation requires 0 bits because we don't need to record anything. The entirety of d can be reconstructed.[4] If $H(d) = \log_2 26$, then to represent a string of length n would require $n \log_2 26$ bits, i.e., each letter can be represented using (on average) $\log_2 26 \approx 4.7$ bits. This also means that the Huffman code tree for d will have a depth of $\lceil \log_2 26 \rceil = 5$, which means that the tree is the most balanced possible that still has 26 leaf nodes. Note that $H(d) = 0$ and $H(d) = \log_2 |S|$ are the two extremes of entropy, representing perfectly predictable and perfectly random data. As an example, perfectly random data of length n built out of $S = \{0, 1\}$ will have an entropy of $\log_2 |\{0, 1\}| = \log_2 2 = 1$, meaning that its most compact encoding uses at least n bits. Thus, perfectly random binary data cannot be compressed!

We can, in fact, verify this in practice. Let us try to compress two files, one containing the first megabyte from the collected works of Shakespeare, which is not very random because it uses natural language with many predictable patterns of characters, and the other a file of the same size built using a random number generator. We can then try to compress both files using the `zip` program, which is an implementation of a popular compression algorithm known as *Deflate*. Here is what we observe.

```
$ head -c 1048576 shakespeare.txt | zip > shakes.zip
  adding: - (deflated 63%)

$ head -c 1048576 /dev/urandom | zip > rand.zip
  adding: - (deflated 0%)

$ ls -l *.zip
-rw-r--r-- 1 kaustuv kaustuv 1048932 May  2 00:48 rand.zip
-rw-r--r-- 1 kaustuv kaustuv  386963 May  2 00:48 shakes.zip
```

The Deflate algorithm was able to remove 63% of bytes from the Shakespeare file, but failed to compress the file of random bytes at all. (In fact, it made the size slightly larger.)

7.4 USING ENTROPY TO CLASSIFY THINGS: DECISION TREES

Data entropy has a few practical uses in algorithms and data structures. One classic example is *decision trees*, which are trees that can be used to *classify* data. The internal nodes of the decision tree contain the labels of particular attributes or *features* of the data being classified, while the edges of the tree are labeled by the values that the features are allowed to have. The leaf nodes of a decision tree contain *classifications*.

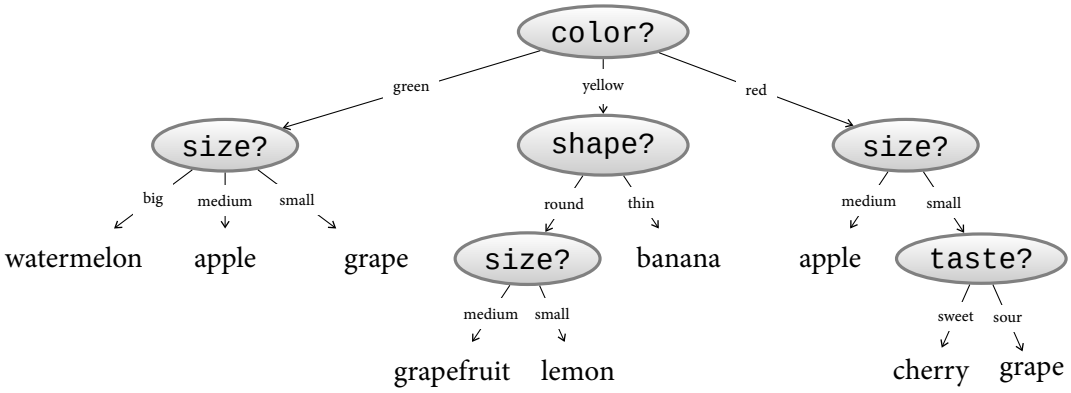
[4] This assumes that we know what $|d|$ is.

To make this concrete, let us consider the example of classifying fruit. Imagine that you are building a robot that has sensors that are capable of detecting the color, shape, size, smell, etc. of fruit, using which the robot has to make a determination of what fruit is placed in front of it. Each of these aspects of fruit is a feature of the fruit, while the identity of the fruit is its classification. In concrete terms, let us say that we want to classify the following kinds of fruit: apples, bananas, grapes, lemons, grapefruit, cherries, and watermelons, using the features of: color, size, shape, and taste. Here is a table that lists all possible combinations that we want to classify.

		feature		
color	size	shape	taste	**classification**
green	medium	round	sweet	apple
yellow	medium	thin	sweet	banana
red	small	round	sour	grape
yellow	small	round	sour	lemon
red	medium	round	sweet	apple
yellow	medium	round	sour	grapefruit
green	small	round	sour	grape
red	small	round	sweet	cherry
green	small	round	sweet	grape
green	big	round	sweet	watermelon

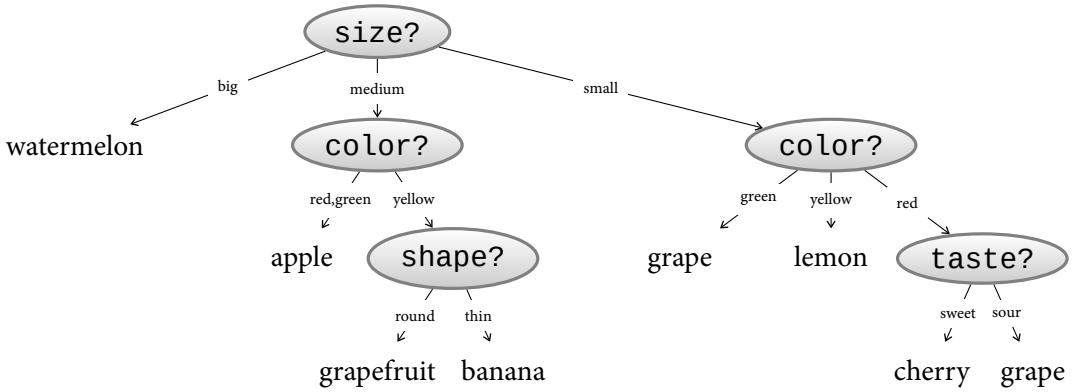
Note that not all combinations of features lead to classifications. For instance, this table is not able to classify an object that is big, red, thin, and sour.

USING A DECISION TREE An example of a decision tree for this feature table is shown below.



To use this decision tree to classify an object based on its features, start at the root of the tree. The feature at the root is color, so the value of the color feature of the object being classified determins which part of the decision tree to look at next. Suppose that the color was yellow; this would result in picking the middle subtree. The feature at its root is shape; if the shape is thin, we can directly produce the classification of banana, since the only yellow and thin objects in our original feature table was a banana. If not, we next look at the size feature, which selects grapefruit or lemon as the classifications, for the sizes of medium and big respectively. If the feature values don't have a corresponding label, then we return a classification of "unknown"; for example, this tree does not give a classification for a yellow, round, big thing.

WHAT MAKES A GOOD DECISION TREE? It turns out that the decision tree that corresponds to a given feature table is not unique. As an example, the above feature table could also correspond to the decision tree shown below, which not only has a different internal layout, but also can potentially give a classification by looking at fewer features (for the case of watermelons). Note also that the edge that leads from the color feature test for a medium size can match both values "red" and "green", so in general we can have multiple values per edge (or, alternately, multiple edges with one label each).



The obvious question to ask is: which tree is better? This question can be answered in a number of ways. An information theoretic way to answer this question is to look at the entropy of the feature table that results from the various choices for the feature at the root of the tree. To keep things simple, assume that our features are drawn from a finite feature set $\mathcal{F}$, and for each $f \in \mathcal{F}$ there is a finite set of values, V_f of that feature.[5] We will assume that there is a designated feature $\star \in \mathcal{F}$, called the *classifying feature* (aka *output feature*) and $V_\star$ is the set of *classifications*. A *dataset* D is a collection of mappings from each feature $f \in \mathcal{F}$ to a value $v \in V_f$; you can think of it as a list of dictionaries, where each dictionary has $\mathcal{F}$ for its keys and each key f is mapped to a value in V_f.

Given such a dataset D, we can compute its entropy $H(D)$ as follows:

$$H(D) \quad = \quad -\sum_{v \in V_\star} p(v, D) \log_2 p(v, D)$$

where $p(v, D)$ is the fraction of the entries in D for which the feature $\star$ has value v, i.e., the fraction of the entries in D with classification v. In addition, given a feature $f \in \mathcal{F} \setminus \{\star\}$, we can *partition* the dataset D based on the value of the feature f; in particular, we define the subset $D[f := v]$ as follows:

$$D[f := v] \quad = \quad \{d \in D : d[f] = v\}.$$

We use this to compute the *weighted entropy* $H(D; f)$ of the dataset as follows:

$$H(D; f) \quad = \quad \sum_{v \in V_f} \frac{|D[f := v]|}{|D|} H(D[f := v]).$$

Note that there is no $-$ sign in the definition of $H(D; f)$.

This brings us to the key concept for comparing decision trees: *information gain*. Given a feature $f \in \mathcal{F}$, we say that the information gain from f, written $I(D; f)$, is the value:

$$I(D; f) \quad = \quad H(D) - H(D; f).$$

[5] The restriction to finite value sets is just to keep things simple for the purposes of this course. In general decision trees can have infinite value sets, and the edges just use some partial partitioning of the values for their labels.

Given two decision trees for D, we compare them as follows: look at the feature at the roots of the two trees, and the information gain from them. The tree that has a larger information gain at the root is better; and if the information gain is identical, then we examine the subtrees one by one.[6]

Building a decision tree: the ID3 algorithm We now have all the ingredients for the algorithm that builds a good decision tree based on information gain. The algorithm was first proposed by the computer scientist Ross Quinlan, and goes by the name ID3. To recap, we have the following information:

- A finite set of features $\mathcal{F}$ with a designated $\star \in \mathcal{F}$
- For each feature $f \in \mathcal{F}$, a finite set V_f of possible values
- A dataset D which is a collection of mappings from $f \in \mathcal{F}$ to V_f.

ID3 is a recursive algorithm that takes D as a parameter and proceeds as follows.

1. If $H(D) = 0$ then all the elements $d \in D$ have the same classification (do you see why?). So, create a leaf of the decision tree with classification $d[\star]$ for some $d \in D$ and finish.
2. Otherwise, find a $f \in \mathcal{F} \setminus \{\star\}$ that maximizes $I(D; f)$. This f will become the label of the root of the tree. There will be $|V_f|$ edges to children that come out of this root node. Each such edge is labeled with a unique $v \in V_f$, and it links to a decision tree for $D[f := v]$.

Keen readers may have noticed that this algorithm is a kind of steepest ascent hill climbing procedure. At each step of the algorithm, it considers a node of the decision tree and a neighborhood determined by the features and their corresponding information gains. It then picks the feature with the maximum information gain. Like all greedy algorithms based on local search, this is not guaranteed to find the best possible tree. In fact, ID3 tends to build trees that are overly complex, with unnecessarily large depths.

Empirical evaluation: training and validation The goal of the ID3 algorithm is to build a tree that classifies a given datum based on the dataset it has "trained" on, i.e., that has been fed to it during the algorithm above. Practical datasets tend to have a lot of repetition in them, so a standard technique for evaluating the quality of the decision tree learning procedure is to split a given dataset into two portions: a *training* set and a *test* (or *validation*) set. The training set is used to build the tree, and then the quality of the tree is computed by running it on the validation set and seeing what fraction of them the decision tree classifies correctly.

7.5 EXERCISES

1. Implement heaps as defined in Section 7.1.2 as the class `Heap` instead of as independent functions.

2. Given the Huffman code tree shown in Section 7.2, compute:
   - The encoding of `'ACECHEETAH'`
   - The decoding of `'\x00\xa1\xb7\x5b\x3b'`

3. Write a function `create_huffman_dict(bs)` that first creates a Huffman code tree from `bs` and then uses it to extract the Huffman code for each byte in `bs`.

[6] To make this ordering precise, we have to invoke a *multiset ordering* of the feature sets, which is probably something you have not yet encountered in your mathematics course-work. So, we will leave a precise mathematical definition of the relative ordering of decision trees to a more advanced course.

```
>>> create_huffman_dict('A DEAD DAD CEDED A BAD BABE A BEADED ABACA BED')
{' ': '00', 'D': '01', 'A': '10', 'E': '110', 'C': '1110', 'B': '1111'}
```

You may find it worthwhile to write a function that performs an inorder traversal of the Huffman code tree to extract the mapping from leaves to the paths from the root encoded therein.

4. Use the previous function to write a function `huffman_encode(bs)` that returns a 3-tuple of the Huffman coding dictionary, the corresponding encoding of the string `bs`, and the amount of padding to the left with 0 bits that was required to make it a multiple of 8-bits.

5. The ID3 algorithm described in Section 7.4 works with value sets V_f that are finite. Figure out how you would extend it to discrete infinite sets, such as the set of integers.

6. Extend the ID3 algorithm to handle the case where the dataset contains incomplete information: for instance, imagine that there is a feature $f \in \mathcal{F} \setminus \{\star\}$ for which not every element of the dataset assigns a value. How would you modify the definitions of $H(D; f)$ and $I(D; f)$ to account for this?

8 BIO-INSPIRED COMPUTING

One of the most fundamental insights of the 20th century is that a lot of phenomena that are exhibited in naturally occurring organisms is inherently computational. Some famous examples:

- The structure of ferns, mollusc shells, coral reefs, etc. is generated by simple recursive procedures reminiscent of the Fibonacci recurrence.
- Nucleic acid polymers (DNA and RNA) contain digital encodings of instructions to build proteins, which in turn are used to build the components of cells and to organize cells into organs and organisms. These include instructions for proteins that copy and modify the instructions themselves, which forms the basis of self-replication, mutation, and evolution by natural selection.
- The nervous systems of most multi-cellular organisms consist of simple computational nodes (neurons) with non-linear interconnections (axons and dendrites). A brain is therefore a kind of computational network.

These observations have led to the field of *bio-inspired computing*, which is an umbrella category of techniques and algorithms that attempt to mimic or model biological phenomena. In this chapter we will go into detail of one such technique, genetic algorithms, and briefly describe a few other techniques that are beyond the scope of this introductory course on computer science.

8.1 GENETIC ALGORITHMS

A *genetic algorithm* (GA) is an algorithm that mimics the process of evolution by natural selection to gradually improve a *population* of *candidate solutions* to a particular problem. The key distinguishing feature of GAs is that each such candidate solution is conceptualized as a *genome*, and in order to create new candidate solutions we can perform the following operations inspired by biological reproduction:

- *Recombination*: given one or more parent genomes, child genomes are created from it by blending together portions of the parent genomes. The most common recombination technique is *crossover*, wherein the parent genomes are cut at the a number of points (at least one), and then the constituent parts are intermixed to create child genomes.
- *Mutation*: child genomes that are produced by recombination may be modified by randomly altering parts of the genome. This simulates the natural process of mutation in DNA that can be caused by copying errors during the ordinary process of replication of DNA strands, or by external mutagenic factors such as toxic chemicals or γ-radiation. In the context of GAs, mutation amounts to modifying the content of a part of a genome at a certain *rate*.

EXAMPLE: THE KNAPSACK PROBLEM We will illustrate GAs by means of the example of the *knapsack problem*, which is one of the classic problems of computer science. It is known to be NP-complete, meaning that the best algorithms that we currently know for solving it are no better than brute force solution, taking exponential time. We are going to develop a GA that gives a *perfect or nearly perfect* answer to the problem using only a fixed amount of computation.

The knapsack problem is as follows: imagine that we have a knapsack that can accommodate items up to a maximum non-negative weight $w_{\max} \in \mathbb{R}^+$. We are given n items $I = \{x_0, x_1, \ldots, x_n\}$ such that each

item $x \in I$ has *weight* $w_x \in \mathbb{R}^+$ and *value* $v_x \in \mathbb{R}^+$. The goal is to find a subset of the items whose total weight is $\leq w_{\max}$ and whose total value is as large as possible. That is, the goal is to compute:

$$\operatorname*{argmax}_{\substack{J \subseteq I, \\ \left(\sum_{x \in J} w_x\right) \leq w_{\max}}} \left(\sum_{x \in J} v_x \right).$$

To illustrate the problem, imagine that $I = \{0, 1, \ldots, 6\}$, $w_{\max} = 9$, and w and v are both represented as Python dictionaries:

```
w = {0: 2, 1: 3, 2: 6, 3: 7, 4: 5, 5: 9, 6: 4},
v = {0: 6, 1: 5, 2: 8, 3: 9, 4: 6, 5: 7, 6: 3}.
```

The sole solution to this problem is the subset $\{0, 3\}$ for which the total weight is $w[0] + w[3] = 2 + 7 = 9 \leq w_{\max}$ and total value is $v[0] + v[3] = 6 + 9 = 15$. Any other subset of items either has total weight $> w_{\max}$ or total value < 15.

GENETIC REPRESENTATION A *candidate solution* (also known as an *individual*) in a GA is an answer that fits many but not necessarily all the parameters of the expected answer; in particular, a candidate solution may violate some desired invariants or be a non-optimal answer. For the example of the knapsack problem above, all subset of the set I of items is a candidate solution, representing the subset of items that we place inside the knapsack. Some of these subsets could have a total weight above $w_{\max}$, which violates our invariant, while other subsets could have a value less than the best achievable value. That is fine.

The first task is to represent such candidate solutions as *genomes*, which is known as the *genetic representation* problem. In rough biological terms, a genome is a collection of *genes*, where each gene encodes a particular value for a given feature.[1] Thus you can see each genome as the instructions to build an individual. In our particular example of the Knapsack problem, we can think of there being n features, one per item of I that determines whether that item is present or absent in a candidate solution to the knapsack problem. Thus, the genome in our case is a mapping g from I to Booleans, such that for each $x \in I$, if $g[x]$ is `True` then x is in the candidate solution, and if $g[x]$ is `False` then x is not in the candidate solution. From the mapping we can extract the actual individual, which is the subset of I that is in the knapsack, by just filtering out those elements $x \in I$ for which $g[x] =$ `True`.

For example, imagine that we wanted to represent the individual/candidate solution $\{1, 3, 6\}$ from the example problem above. We could write out its genome g as a Python dictionary.

```
g = {0: False, 1: True, 2: False, 3: True, 4: False, 5: False, 6: True}
```

We were not required to use a dictionary to represent the genome. Since the set of items I in this example is just `set(range(6))`, we could have used a *list* or a *tuple* of Booleans, where the ith element of the list or tuple would determine whether $i \in I$ was in the candidate solution or not. We could even use a compact bitset representation like in Chapter 2.5.

FITNESS Each individual/candidate solution has an associated quality called its *fitness* (sometimes also called *goodness*) that can be used to compare different individuals in search of a best individual. In our knapsack problem example, the fitness of an individual is just the sum of the values of the items that

[1] Always keep in mind that this is a drastic simplification of actual biology.

are placed in the subset of I by the genome mapping. However, note that we have defined our genetic representation to allow for genomes that could potentially violate the maximum weight restriction; thus, any such candidate solution should be assigned a very low fitness score such as 0.

Here is how the fitness computation looks like in Python:

```
def fitness(weights, values, max_weight, gnm):
    """Compute the fitness of the individual with genome `gnm`."""
    # note: `weights` is w, `values` is v, and `max_weight` is w_max
    total_weight = sum([weights[x] for x in gnm if gnm[x]])
    total_value = sum([values[x] for x in gnm if gnm[x]])
    if total_weight <= max_weight:
        return total_value
    # otherwise return a very low fitness
    return 0
```

Recombination by crossover Given two individuals—the *parents*—we can make two new individuals from them—the *children*—by blending together the genomes of the parents. This is known as recombination, and the most popular variety is *crossover*, which is in fact also observed in sexual reproduction.[2] Each crossover between two parents p_1 and p_2 produces two children c_1 and c_2; moreover, c_1 contains some of the genes of p_1 and the complementary set of genes of p_2, and vice versa for c_2.

To express this in Python, we walk through all the genes of both parents one by one, and for each gene we flip a fair coin to determine which child gets which gene. This allows for a perfect recombination of the parent genomes into the child genomes.

```
def crossover(p1, p2):
    """Perform a uniform crossover of the parent genomes `p1` and `p2`."""
    # we assume that p1 and p2 are dictionaries with the same keys
    c1, c2 = {}, {}
    for k in p1:
        if random.randint(0, 1) == 0:      # toss a fair coin
            c1[k], c2[k] = p1[k], p2[k]
        else:
            c1[k], c2[k] = p2[k], p1[k]    # note that p1/p2 is swapped
    return (c1, c2)
```

Note that this definition of `crossover()` is quite versatile since it is largely independent of what specific keys and values are present in a genome. It certainly works for our knapsack problem example, but it would also work for pretty much any other genetic representation in the form of dictionaries.

If the genomes were to be represented not as dictionaries but as lists or tuples, then the crossover function would be a bit more cumbersome to implement. In *single point crossover*, an index `k` is in both genomes is uniformly selected, and then the children are built from a cross-pairing of the two parts of each genome like so:

```
    k = random.randint(0, len(p1))
    c1 = p1[:k] + p2[k:]
    c2 = p2[:k] + p1[k:]
```

[2] Technically, in biology, crossover happens during meiosis—the formation of gametes—where the genetic material of the parents of an individual are recombined to form half the chromosomes that individual will pass on to their own children. Thus, each child contains a blend of the genetic material from their *grandparents*, not their immediate parents. Thus, the GA perspective on recombination is a not a perfect simulation of biology.

Unfortunately this has the effect that the first and the last genes of each genome will be found together in only the trivial case where the children genomes are perfect copies of the parent genomes. (Do you see why?) To fix this, one generally needs to implement *multi-point crossover* by finding several indexes to cut up a genome-qua-list at, and then interleaving the parts.

Mutation The crossover operation creates children genomes from parent genomes by blending the genomes together, but it does not invent wholly new genes. Each gene in each child comes from either the first or the second parent. This means that if crossover were the only phenomenon in play, then the fitness of every child, and indeed every genome, is bounded by the diversity that is present in the initial population of individuals. To break out of this genetic bound, GAs often mimic another important feature of biological evolution: *mutation*.

Every mutation event operates on (the genome of) an individual by making a random alteration to the genome. This alteration can take several forms:

- The value of one of the genes can be randomly altered. For instance, in the knapsack example, a `False` value can be changed to a `True`, or a `True` to a `False`.
- The value of one of the genes may be changed to be identical to that of a different gene with compatible values.
- In cases where a genome is represented not as a dictionary but as a tuple or a list, a given gene can get deleted or duplicated, or the relative order of two or more genes can be changed (e.g., swapped).

In fact, many other kinds of mutation are possible, and new types of mutation are easy to invent.

For our given knapsack problem example with genomes represented as dictionaries mapping keys to Booleans, we can write the following function for mutations of the first kind.

```
def mutate(g):
    """Perform a single flip of a gene in `g`"""
    k = random.choice(list(g.keys()))   # select a random key
    g[k] = not g[k]                     # flip its value
```

Selection A *population* is a collection (list, set, etc.) of individuals. The core loop of a GA is to take one such *parent population* and then build from it a *child population* using the crossover and mutation operators discussed earlier. To create children, two parents are *selected* from the parent population for breeding. Before we describe how the breeding itself occurs, it is important to discuss how to write this selection function.

As with many aspects of GAs, different instances will have different selection functions. A common selection function is *selection of the fittest*, where in each round two fittest genomes in the parent population are selected for breeding children. These parents are then removed from the parent pool and then the next round of selection happens with the next two fittest parents. This continues until there are 0 or 1 parents left in the parent population. The code for this for the knapsack problem is shown in the `select_fittest_first()` function in Figure 8.1, which simply returns the list of parents sorted in decreasing order of fitness.

The problem with selecting the fittest pair of parents always is that it gives a very high bias to fitness which is not how biology works in practice. Indeed, it is very unusual for an entire population of creatures to self-organize according to fitness and then to pair off according to their ranks. Rather, there is an element of chance in forming breeding couples where the fitness is used as a bias but not a hard guarantee. The higher the fitness of a creature, the more likely it is to be selected to form a member of a breeding

```
def select_fittest_first(weights, values, max_weight, parents):
    """Return a list of parent genomes using a fittest-first approach"""
    # make a copy of the parents list
    parents = parents[:]    # or use copy.deepcopy()
    # sort the parents in *decreasing* order of fitness
    parents.sort(reverse=True,
                 key=lambda gnm: fitness(weights, values, max_weight, gnm))
    return parents


def select_fittest_ranked(weights, values, max_weight, parents):
    """Return a list of parent genomes using ranked selection"""
    # make a copy of the parents list
    parents = parents[:]    # or use copy.deepcopy()
    # sort the parents in *increasing* order of fitness
    parents.sort(key=lambda gnm: fitness(weights, values, max_weight, parents))
    # total fitness of the parents
    total_fit = sum([fitness(weights, values, max_weight, gnm) \
                     for gnm in parents])
    sels = []
    while parents:
        cutoff = random.uniform(0, total_fit)
        # assuming the parents are laid end-to-end where each parent
        # takes up space equal to its fitness, find which parent
        # is at the cutoff, starting at the end
        k = len(parents) - 1
        while k > 0:
            cutoff -= fitness(weights, values, max_weight, parents[k])
            if cutoff < 0: break
            k -= 1
        sels.append(parents[k])
        total_fit -= fitness(weights, values, max_weight, parents[k])
        del parents[k]
    return sels
```

Figure 8.1: Several selection functions

pair. In order to implement this kind of selection, we use *ranked choice* (seen in Chapter 3.1). This is implemented in the `select_fittest_ranked()` function in Figure 8.1.

PUTTING IT TOGETHER Given all the ingredients—fitness, crossover, mutation, and selection—we can finally assemble the full genetic algorithm that performs evolution by natural selection to improve the maximum fitness of a population. An initial population can be built using completely random genomes. We order the population based on one of the selection functions shown in Figure 8.1. Then, to build the next generation of individuals, we repeatedly pick pairs of parents from this list and use `crossover()` to make two children. Each child then undergoes mutation with a given mutation chance. At the end, when there are no more pairs to be formed, we compare the fittest individual in the parent and child populations, which we call the fitness of the population itself. If the child population is at least as fit as the parent population, we keep going; otherwise, if the child population has a strictly lower fitness, we terminate the algorithm and return the fittest member of the parent population as the output of the GA. To guarantee that we eventually find an answer, we may also bound the maximum number of generations to consider.

Here it is in Python for the knapsack problem

```
def knapsack_ga(weights, values, max_weight):
    """Return a genome with fitness as high as possible using GA"""
    # the parameters of the algorithm; can be tweaked
    pop_size = 100
    mutation_rate = 0.03
    max_generations = 10_000
    # initial population is random:
    parents = [ {k: random.choice(True, False) for k in weights}
                for _ in range(pop_size) ]
    fittest = max(parents,
                  key=lambda gnm: fitness(weights, values, max_weight, gnm))
    for _ in range(max_generations):
        sels = select_fittest_ranked(weights, values, max_weight, parents)
        kids = []
        while len(sels) > 2:
            p1, p2 = sels[0], sels[1]
            del sels[:2]
            c1, c2 = crossover(p1, p2)
            if random.random() < mutation_rate: mutate(c1)
            if random.random() < mutation_rate: mutate(c2)
            kids.extend([c1, c2])
        fittest_kid = max(kids,
                          key=lambda gnm: fitness(weights, values, max_weight, gnm))
        if fitness(weights, values, max_weight, fittest) > \
           fitness(weights, values, max_weight, fittest_kid): break
        parents = kids
        fittest = fittest_kid
    return fittest
```

9 SECURITY AND DATA INTEGRITY

Very sophisticated computing devices not only reside in our homes, schools, businesses, and governments, but increasingly also travel with us in our cars, in our pockets, and on our person. This ubiquity has made many civilizational advances possible: we are always just a few keystrokes away from anyone we have ever met, we always know exactly where we are on Earth, and we can participate in real time in global conversations about practically any subject. But, this impressive connectedness comes with a similarly impressively huge problem: it has never been easier for criminals, shady businesses, nosy governments, or other nefarious agents to track, monitor, and impersonate you. How would you protect yourself and your data from such threats, especially when the threats may come from sources that have the wealth of entire nations backing them up? The answer, unsurprisingly enough, comes from mathematics, specifically the theory of *one-way functions* that forms the basis of modern cryptography.

9.1 HASHING

ONE WAY FUNCTION The basis of all modern cryptography done with the use of computers is to define functions that can translate input bit sequences to output bit sequences with certain properties. Given such a function $f : \mathbb{N}_2^* \to \mathbb{N}_2^*$, we say that it has a *pseudo-inverse*[1] $g : \mathbb{N}_2^* \to \mathbb{N}_2^*$ if: for every $x \in \mathbb{N}_2^*$ it is the case that $f(g(f(x))) = f(x)$. The function f is said to be a *one-way function* if it has polynomial time complexity—i.e., there is an $n \in \mathbb{N}$ such that $f(x)$ will be computed in time $O(|x|^n)$—but it does not have a pseudo-inverse that has polynomial time complexity.[2]

The amazing thing about one-way functions is that even though they form the basis of modern cryptography, we do not know if one-way functions even exist. That is, no one has constructed a function f and proved that it has no polynomial time pseudo-inverse. If such a function can be constructed, then the complexity class P (of all polynomial time programs) would be strictly contained in the complexity class NP (of polynomial time programs that are allowed to make perfect guesses). Finding a one-way function would therefore resolve possibly the most famous open question in computer science, that of whether P is equal to NP. Note that proving P is strictly contained in NP would not imply that one-way functions exist; this is a very common mistake made by people who fail to understand that "A implies B" is not the same as "B implies A".

Despite this theoretical concern, there are a number of functions in practice that have been empirically determined to be *very hard* to (pseudo-)invert. That is to say, we know of functions that, despite the best efforts of some of the top mathematicians and computer scientists who have ever lived, we have not been able to invert. In fact, you have seen such functions already: every (good) *pseudorandom number generator* is inherently a one-way function from its internal state to its output sequence. Knowing just the output of such a generator, you cannot determine its internal state, and hence you cannot predict the rest of the sequence it generates.

[1] Why pseudo-inverses? So that our definition will work even for functions that are not one-to-one. If the function happens to be one-to-one, then g will be a left-inverse of f, i.e., for every x it will be the case that $g(f(x)) = x$.

[2] Strictly speaking, g is even allowed to be a randomized function, i.e., that it has access to a random number generator that it can call a polynomial number of times, and we only require that the probability that $g(f(x))$ will happen to be a x' such that $f(x') = f(x)$ is vanishingly small. In other words, f is a one-way function if it has no pseudo-inverse that has *randomized polynomial time* complexity. Since this is a concept that you haven't yet been formally introduced to, we will leave this precise definition of one-way functions to future courses.

Cryptographic hash functions Among this broader class of functions that are empirically difficult to invert are the *cryptographic hash functions*, which are hash functions that transform input of arbitrary length to an output of a fixed size, i.e., it belongs to the function space $\mathbb{N}_2^* \to \mathbb{N}_2^k$ for a fixed $k \in \mathbb{N}$. Like all functions, a cryptographic hash function is deterministic: it always maps the same input to the same output, no matter when or how many times it is called. In addition, a good cryptographic hash function $h : \mathbb{N}_2^* \to \mathbb{N}_2^k$ has the following properties.

1. Given $x \in \mathbb{N}_2^*$, it takes time $O(|x|^n)$ (for some fixed $n \in \mathbb{N}$) to compute $h(x)$. Ideally, $n \approx 1$. Hash functions that have large n are not particularly useful in resource-intensive applications, though they are not without uses. For instance, if it is important to limit the *rate* at which inputs can be hashed, then having a high n is desirable.[3]
2. Given a $y \in \mathbb{N}_2^k$, it is computationally infeasible to find an $x \in \mathbb{N}_2^*$ such that $h(x) = y$. That is to say, the best way to find such an x would be to try every element of $\mathbb{N}_2^*$ one by one. A hash function that fails to have this property is said to be susceptible to *first preimage attack*. (Often the "first" is dropped from this description.)
3. Given a fixed $x_1 \in \mathbb{N}_2^*$, it is computationally infeasible to find a $x_2 \in \mathbb{N}_2^*$ such that $x_1 \neq x_2$ and $h(x_1) = h(x_2)$. A hash function that lacks this property is said to be susceptible to a *second preimage attack*.
4. It is computationally infeasible to find two $x_1, x_2 \in \mathbb{N}_2^*$ with $x_1 \neq x_2$ for which $h(x_1) = h(x_2)$. This is different from the previous case because both x_1 and x_2 can be freely chosen. A hash function that lacks this property is said to be susceptible to a *collision attack*.
5. Let the *distance between* two elements $x_1, x_2 \in \mathbb{N}_2^*$ be defined to be the number of bits in x_1 that must be flipped in order to get x_2, written as $x_1 \ominus x_2$. A good hash function has the property that $x_1 \ominus x_2$ is not correlated with $h(x_1) \ominus h(x_2)$. That is to say, if we plot the points with coordinates $(x_1 \ominus x_2, h(x_1) \ominus h(x_2))$ for various choices of x_1 and x_2, we would see a uniform distribution of points. Hash functions that have correlated values of $x_1 \ominus x_2$ and $h(x_1) \ominus h(x_2)$ are said to be *leaky*.

The last property, in particular, means that we cannot use similarity of the output of the hash function to learn anything about the inputs. Any hash function that is not susceptible to a certain kind of attack is said be *resistant* to the attack.

Application: provenance One of the principal uses of cryptographic hash functions is to use it to *prove* the *provenance* of data, which is to say, build a digital proof of where the data comes from.

Imagine that you have discovered a secret and you want to announce to the world that you have found it without revealing what the secret is. To be concrete, imagine a game of *secret treasure hunt* where the goal is to secretly discover the location of a treasure. Imagine a player, Carol, who wants to prove to her competitors that she has discovered the location of the treasure, `(13, 42)`, without revealing where the location is, or even that it was she who discovered it. Given a cryptographic hash function `h()`, Carol can (anonymously) publish the value of:

```
h('Carol discovered the treasure location: (13, 42)').
```

Because of first preimage resistance, none of the other players can determine the string that Carol used to produce the hashed value. At the end of the game, when it is time to claim her victory, all she has to do is

[3] This is commonly used to stop *denial of service (DOS) attacks* in the public internet.

to reveal the source string,

```
'Carol discovered the treasure location: (13, 42)'.
```

The rest of the players can quickly verify that Carol is telling the truth by applying `h()` to it. Note that because the source string is fully in Carol's control, she can try to flummox any attempts to brute force her source string by using string that is very hard to guess, such as:

```
'The treasure is at (13, 42). Carol found it. The moon is made of cheese.'.
```

In fact, because the hash function is resistant to leaks, she could just add a random amount of whitespace to her string (shown below as `' '`); it will be just as unguessable.

```
'Carol discovered the treasure location: (13, 42)     '
```

Application: passwords A common way to give access to a resource to only a select group of people is to protect the resource using a *password*. This idea is as ancient as civilization, but until the advent of modern computing passwords were not very secure. Indeed, all that was required to break passwords in the olden days was to eavesdrop on someone using the password. Nowadays it is very rare to the point of irrelevance for passwords to be shared among groups of people. Instead, every person who should have access to a resource is allowed to pick their own individual and secret passwords.

Imagine that Arya wants to log into a computer system with her username, `'arya'`, and her secret password, `'winteriscoming'`. At a very basic level the server could store a big list of mappings from user names to passwords, such as:

```
arya: winteriscoming
bronn: nocure
dany: dragonsdaughter
hodor: hodor
...
```

However, this is not very secure, since anyone on the server who can read this file will know everyone's password and can therefore impersonate anyone. Hence, the server generally does not store the password in *plaintext* form, but rather only stores a hash of the password. For instance, it could use the Python function `crypt.crypt()` for this purpose.

```
>>> import crypt
>>> crypt.crypt('winteriscoming', crypt.METHOD_CRYPT)
'D5ssafc4Q1NNo'
>>> crypt.crypt('dragonsdaughter', crypt.METHOD_CRYPT)
'x1GXxkYjMxcSI'
```

The second argument to `crypt.crypt()` says to use one of the oldest and most insecure hash functions, the *Crypt* function, that used to be popular before the 1970s. The first two characters of the returned string is called the *salt*, which is a random string prepended to the plain password before computing its hash.[4] That is to say, the `crypt.crypt()` function actually generates the random string `'D5'`, and then

[4]The full reasons for using salts is beyond the scope of this course. Briefly, though, one purpose the salt serves is to prevent multiple users with the same passwords from having the same hashed password. More generally, salts prevent brute force attempts to recover passwords from hashed password databases—cracking—using techniques based on pre-computing the hashes of many common words and passwords, such as all the words indexed in an English dictionary or passwords obtained by earlier successful cracking attempts.

hashes the string `'D5winteriscoming'` to get the result `'ssafc4Q1NNo'`. The salt and the hash are then concatenated and printed out. Using this mechanism, we can store the passwords as follows:

```
arya: D5ssafc4Q1NNo
bronn: JNHVMCyjpScz2
dany: x1GXxkYjMxcSI
hodor: 6szHDYlcGZw.E
...
```

The salts (the first two characters) in each of the hashed passwords is arbitrary.

Now, when the server receives the username/password pair `('arya', 'winteriscoming')` from Arya, it will use `crypt.crypt()` to check that the password matches the hash it has stored for the username `'arya'` together with its salt `'D5'`, and on success allow Arya to continue:

```
>>> crypt.crypt('winteriscoming', 'D5')
'D5ssafc4Q1NNo'
```

Meanwhile, no one else who knows just the (salted) hashed string `'D5ssafc4Q1NNo'` but not the plain password `'winteriscoming'` will be able to log in as Arya.

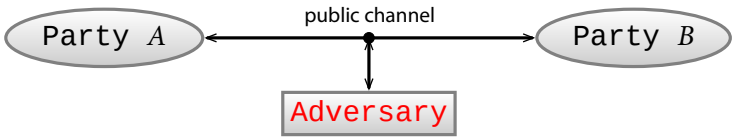
9.2 CRYPTOGRAPHY AND SECURITY GOALS

The cryptographic hash functions in the previous section were, effectively, single parameter functions: given an input, they produce an output that is difficult to invert. We will now generalize this to discuss cryptographic functions that take several inputs, and furthermore discuss functions that allow for inversion in certain restricted settings. Before we dive into definitions and techniques, let us get a high level overview of the field.

Cryptography is the practice and the study of techniques for *secure computation* in the presence of *adversarial* third parties that can:

- *observe* and *record* all the data that is communicated on public channels;
- can prevent the delivery of any particular data or inject its own data, or any recorded data, into the communication stream.

An adversary that merely observes the channel is called a *passive adversary*; all it can do is try to learn secrets that the parties want to keep private. Adversaries that also intercept and inject their own data are called *active adversaries*; such adversaries can try to mislead the parties both to gain private knowledge and to engineer malfunctions. In the simplest case we have two parties, A and B, that are communicating over a public channel such as a connection on the public internet. Unbeknownst to them, their public channel has a third participant, an adversary. We can depict it as follows:



With this setup, cryptography can be used in the service of several *security goals*, of which the following are the most important and well known.

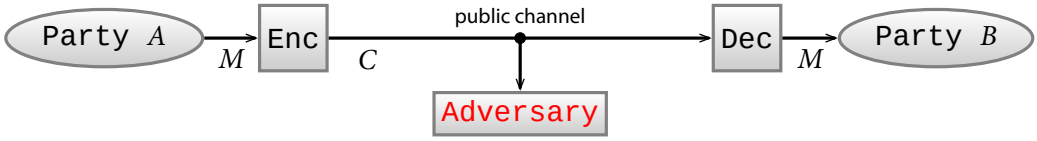
- ***Privacy***: information that is intended to be secret to the parties in the communication (A and B in the above case) should not be learned by the adversary. Imagine, for instance, if A is a person and B is a bank, and the private information is A's bank balance. The adversary must not be able

to determine A's balance by observing or modifying the bytes that are transferred between A and B. Privacy is usually achieved using *encryption*.

- ***Integrity***: information that is sent from one party must arrive at the other party without any undetected modifications. Imagine that A is a voter and B is an election system. The adversary must not be able to modify A's votes. Integrity is achieved using *digital signatures* and *message authentication codes* (MAC).

- ***Authentication***: each member, A and B, of the party must be able to trust that the other member is who they say they are. The adversary must not be able to impersonate any parties. Authentication is also achieved using signatures and MACs, and in some cases using shared secrets.

All cryptographic communication happens with the help of a pair of *encryption* and *decryption* functions, which are lossless encoding/decoding functions of the form you already saw in Chapter 7. We can visualize these in the form of an *encoder* and a *decoder*, shown below as boxes.



Party A that wants to communicate a message M to party B first sends it to the encoder; this message is called the *plaintext*. The communication between the parties to the encoder (or decoder) is not a public channel; usually this is done inside a single program by use of cryptographic libraries. The output of the encoder is an encrypted message C called the *ciphertext*, which is sent on the public channel where the adversary prowls. The decoder receives the ciphertext and then *decrypts* it back into the plaintext, which is then sent via a private channel to the party B.

There are two main kinds of cryptography based on how the encoder and decoders work: *symmetric key* and *asymmetric key* cryptography. They are described below after a brief historical interlude.

A BRIEF HISTORY Although cryptographic techniques have been used since ancient times, the science of cryptography was barely developed until the 20th century. The most classic techniques from antiquity mainly involve the use of *obfuscation*. Communication between the parties could be done in a secret coded language that is presumably unknown to the adversaries who can intercept the communication. In some cases this could involve artifacts that may not be in the possession of the adversary, such as code books or gizmos. Another ancient technique is the use of *substitution ciphers*, famously used by Cæsar himself (which is why they are sometimes known as *Cæsar ciphers*). In this technique, which was primarily used to transmit text, certain letters or letter combinations were replaced with other letters or letter combinations. The simplest Cæsar cipher simply replaces every letter by a letter that occurs a fixed distance, say 3, down the alphabet, so the string `'Carthago delenda est'` may be encoded as `'Fduwkdjr ghoqgd hvw'`.

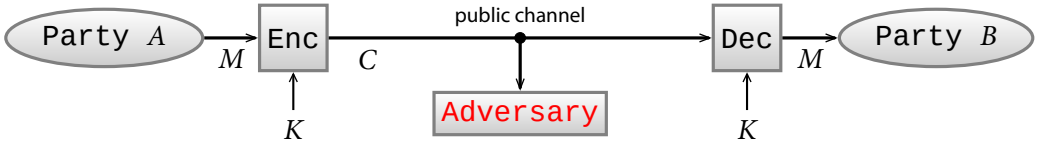
The 20th century saw the use of mechanical and then computational devices to perform the encryption. These machines were far less predictable than substitution ciphers and did not rely on obfuscation. Instead, the encryption made use of properties of modular arithmetic that would be encoded first in gear ratios and later in electronics. The golden age of encryption devices was the World War II, where the German army famously made use of the *Enigma* machines, a typewriter-sized device that used rotors and lights to encode the input entered using its keypad. One such machine was captured by the allied forces from a German U-boat and brought to Bletchley Park, where a team of mathematicians reverse-engineered the

device and was then able to read the military communication of the German army. This period of history has been dramatized a number of times, most recently in the movie *The Imitation Game*.

Since the 1970s, cryptography has transitioned entirely to computer algorithms, and has been turned from an art into a rigorous science starting with the development of information theory by Claude Shannon. All the security concepts laid out at the start of this section were given formal mathematical definitions by Shannon and his successors. One key innovation that came with this formalization is the notion of a *protocol*, which is a mathematical and symbolic description of the *messages* that are communicated between the parties of a communication.

9.3 SYMMETRIC ENCRYPTION

The simplest type of cryptography is *symmetric key* cryptography where both the encoder and the decoder share the same key K which should be kept secret from the adversary.



We write $\mathrm{enc}(M, K) = C$ to mean that the plaintext M encrypts to the ciphertext C using the key K. The defining property for symmetric encryption schemes is the following condition, first formulated by Shannon, which is known as *perfect secrecy*: the adversary knowing only the ciphertext will not be able to learn anything about the plaintext.

Definition 9.1 (Perfect secrecy). *In a symmetric encryption system, knowing only the ciphertext grants no knowledge about the plaintext. Formally, treating M, C, and M as random variables,*

$$P[M \mid C] = P[M],$$

i.e., the plaintext and ciphertext are independent.

As the name implies, perfect secrecy is an *ideal* property. Shannon proved that perfect secrecy can be achieved in only a very specific circumstance: when the key is longer than the plaintext, $|K| \geq |M|$. (In fact, it will be the case that $|K| \geq |C| \geq |M|$.) This means that the shared key K between the two parties must be long enough that the message fits inside the key. Another consequence of Shannon's observation is that perfect secrecy is only achievable if the key K is never used for any message ever again once exhausted; for otherwise we can treat the concatenation of multiple plaintexts as a new longer plaintexts that would violate $|K| \geq |M|$.

ONE TIME PADS It turns out that there is indeed an encryption scheme that achieves perfect secrecy: *one-time pads* (OTP; sometimes called a *Vernon cipher*). Let us fix the key size $|K|$ to be $n \in \mathbb{N}$. An OTP of size n is a uniformly random n-bit binary number, that is an n-bit number where each bit is independently and uniformly selected to be `0` or `1`. Encryption and decryption with such an OTP K are defined as follows:

$$\mathrm{enc}(M, K) = M \wedge K \qquad \text{and} \qquad \mathrm{dec}(C, K) = C \wedge K.$$

where $\wedge$ is the usual exclusive or (xor) operator. Note that $\wedge$ is self-canceling, has `0` as its identity element, and is associative, so:

$$\mathrm{dec}(\mathrm{enc}(M, K), K) = (M \wedge K) \wedge K = M \wedge (K \wedge K) = M \wedge \mathtt{0} = M.$$

Why is this perfectly secure? Well, since K is uniformly selected from the entire space of 2^n possible keys, it must be the case that $P[K = k] = 2^{-n}$ for any bit pattern k. Take the ith bit of the plaintext, M_i. During encryption, to produce the corresponding ith bit of the ciphertext C_i, we take the xor of M_i with the ith bit of the key, K_i. But, this latter value is `1` if an only if $K_i \neq M_i$ and since each bit of K was uniformly selected, by definition the chance that $K_i \neq M_i$ is the same as that of $K_i = M_i$, both being $1/2$. Therefore, $P[C_i = \texttt{1}] = P[K_i \neq M_i] = 1/2 = P[K_i = M_i] = P[C_i = \texttt{0}]$, and hence $P[C = c] = 2^{-n}$ for any bit pattern c. Clearly this is independent of the value of $P[M = m]$, and hence C and M are independent variables, i.e., the encryption has perfect secrecy.

Intuitively, given a ciphertext, every text of the same length is equally likely to have been the plaintext, since the key is itself uniformly selected, and each key maps a ciphertext to a unique plaintext in the decryption function.

OTHER SYMMETRIC SCHEMES While the OTP encryption scheme is simple, OTP keys are very long and not reusable, which makes the scheme very cumbersome in practice. In particular, since keys have to be shared and are not reusable, the two parties must plan well in advance for the total length of their communication. This may be doable in extraordinary circumstances where perfect secrecy is paramount, such as delivering the launch instructions to nuclear missiles in the event of a nuclear war, but in most scenarios we would need smaller and reusable keys. By Shannon's theorem, this would mean that we give up perfect secrecy in favor of better usability.

Covering the theory of particular encryption schemes is beyond the scope of this course, but here is how one might use the AES cipher from Python3. First, we have to install the `pycrypto` library in the standard way.

```
$ python3 pip install --user pycrypto
```

Here is a simple example of the use of one of the available symmetric encryption schemes, AES, for which the key size must be 16 bytes long and the plaintext needs to be a multiple of 16 bytes long.

```
>>> from Crypto.Cipher import AES
>>> key = b'Sixteen byte key'
>>> aesc = AES.new(key, AES.MODE_ECB)
>>> plaintext = b'Attack at dawn'
>>> ciphertext = aesc.encrypt(plaintext)
>>> ciphertext
b'\xdc\n&\xf5\xbb\xf2\xa3\x12S\xd3<&\xe7\x9d\xca\x1d'
>>> aesc.decrypt(ciphertext)
b'Attack at dawn!!'
```

Note that we are using *byte strings* instead of ordinary Python strings here, denoted with a `b` in front of the strings. A byte string is like a regular `str` object, except it is guaranteed to hold bytes instead of Unicode characters. The kth byte in a byte string can be accessed in $O(1)$ time, as opposed to in $O(k)$ for strings (since a single Unicode character may take between 1 and 4 bytes of space in memory).

9.4 ASYMMETRIC KEY CRYPTOGRAPHY

Symmetric encryption schemes are some of the fastest encryption schemes, but they suffer from the fact that the key is a shared secret. If that secret were to somehow leak to the adversary, then the game is up. The adversary can easily pretend to be the party B and hold an encrypted conversation with A without A being any the wiser. Thus, to maintain secrecy, the shared secret must be routinely refreshed, which

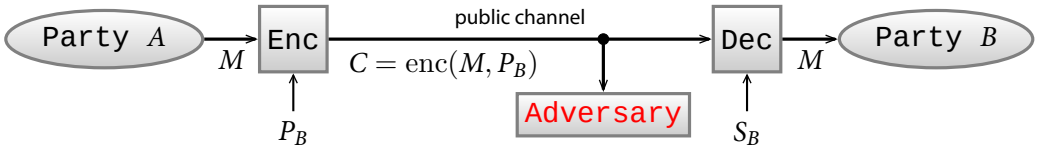
requires the parties to physically meet (for, if they use a communication channel to share the key, then the adversary may sniff *that*).

To avoid these issues, we use *asymmetric key encryption* (also known as *public key encryption* or PKE). In a PKE scheme, each participant (called a *principal*) in the conversation generates a *pair* of keys at the same time. One of the keys in the pair is a *secret key* that the principal keeps completely private, i.e., sharing it with no one else. The other key in the pair is a *public key* that the principal openly publishes anywhere information about the principal can be found, such as a web-site or a online registry of public keys.

When A wants to send a message M to B, they encrypt the message using B's public key, P_B, which they find from wherever B published their public key. Then, when B receives the encrypted message $\mathrm{enc}(M, P_B)$, they use their secret key S_B to decrypt it, so we require that:

$$\mathrm{dec}(\mathrm{enc}(M, P_B), S_B) = M.$$

The scheme looks as follows.



We can extend our notion of perfect secrecy to asymmetric key encryption as follows: the adversary must not be able to learn anything about the plaintext M knowing the ciphertext C and the public key P_B, but not knowing the secret key S_B. The first and most well known asymmetric key encryption scheme is *RSA*, which is based on the prime factorization problem that is known to be computationally difficult. Another famous scheme is *ElGamal* that exploits the difficulty of computing discrete logarithms.

EXAMPLE: RSA The RSA algorithm is named for the initials of its inventors, Rivest, Shamir, and Adleman. To build a (public, private) RSA key pair, we follow the following procedure.

1. Start by finding two large and distinct prime numbers $p, q \in \mathbb{N}$.
2. then compute $m = pq$ (called the *modulus*) and $\lambda = \mathrm{lcm}(p-1, q-1)$.
3. Choose an $e \in \mathbb{N}$ (called the *encrypting exponent*) such that $1 < e < \lambda$ and $e = 1 (\mathrm{mod}\, \lambda)$, i.e., e and λ are coprime.
4. The pair (m, e) forms the public key.
5. Compute $d = e^{-1} (\mathrm{mod}\, \lambda)$ using the extended Euclidean algorithm.
6. The number d forms the secret key.

For example, if $p = 53$ and $q = 97$, then $m = 5141$ and $\lambda = 1248$. We can pick $e = 7$ as our encrypting exponent since 7 is coprime with 1248. Thus one possible public key is $(5141, 7)$. To compute d we use the `mulinv()` function shown in Figure 9.1 to get 535. Indeed, $535 \times 7 = 3745 = 1 (\mathrm{mod}\, 1248)$.

1. Represent the message M as a natural number $< m$. This can be done by interpreting the sequence of bytes in M as the byte pattern of a natural number. If M is too large in this interpretation, divide M into several parts and encrypt each part separately.
2. To encrypt M using the public key (m, e), compute $C = M^e (\mathrm{mod}\, m)$.
3. To decrypt C using the private key d, compute $M' = C^d (\mathrm{mod}\, m)$.

By our choice of e and d, we have $(M^e)^d = M^{ed} = M^1 = M (\mathrm{mod}\, m)$. The algorithm is computationally secure because given $m = pq$, it is very hard to find p and q and hence to find λ and d.

```python
def xgcd(a, b):
    """return (g, x, y) such that a*x + b*y = g = gcd(a, b)"""
    x0, x1, y0, y1 = 0, 1, 1, 0
    while a != 0:
        q, b, a = b // a, a, b % a
        y0, y1 = y1, y0 - q * y1
        x0, x1 = x1, x0 - q * x1
    return b, x0, y0

def mulinv(a, b):
    """return x such that (a * x) % b == 1"""
    g, x, _ = xgcd(a, b)
    assert g == 1
    return x % b
```

Figure 9.1: Extended Euclidean algorithm and modular multiplicative inverses

QUALITATIVE ANALYSIS Asymmetric key encryption schemes have a lot of benefits.

- The keys are stable: they can be maintained over a period of years to decades without any significant decrease in the secrecy that is achieved.
- Since there are no shared secrets, such schemes are much less vulnerable to breaches of trust.
- Furthermore, each key-pair can be independently generated, requiring no synchronizing with the other parties in a communication.

On the other hand, asymmetric schemes are not as performant as symmetric encryption schemes because of their reliance on computationally complex operations such as exponentiation, even though this can be mitigated somewhat using specialized algorithms for modular exponentiation. When the speed of encryption in the communication channel is a bottleneck, it is usual to use asymmetric encryption to do a key exchange for a new symmetric key, which is then used for the remainder of the communication using symmetric encryption. The adversary cannot eavesdrop on the initial key-exchange and hence cannot break the subsequent symmetrically encrypted communication.

Another issue with asymmetric key encryption is that the key sizes need to be large in order to have good secrecy. If our primes p and q are too small, then $m = pq$ can be factorized by brute force. Given our current estimate of humanity's engineering abilities, it is reasonable to believe that a brute force computation that would use fewer than 2^{64} basic operations is feasible (especially for governments or large corporations), while 2^{80} operations would take on the order of 100 years. With these estimates, it is standard to use the following key sizes for secure encryption (aka "strong encryption"):

- Symmetric encryption: a minimum key length of 128 bits is recommended, which would take on the order of 2^{128} operations to brute force.
- Asymmetric encryption: for RSA, a minimum key length of 1536 bits is recommended, and 2048 bits is common.

INDEX