Solution: Assume that the bandwidth from the lab machine is a limiting factor. It is reasonable to first do trivial optimizations, such as combining the articles into a single file and compressing this file.

Opening 1000 connections from the lab server to the 1000 machines in the data center and transferring the latest news articles is not feasible since the total data transferred will be approximately 100 gigabytes (without compression).

Since the bandwidth between machines in a data center is very high, we can copy the file from the lab machine to a single machine in the data center and have the machines in the data center complete the copy. Instead of having just one machine serve the file to the remaining 999 machines, we can have each machine that has received the file initiate copies to the machines that have not yet received the file. In theory, this leads to an exponential reduction in the time taken to do the copy.

Several additional issues have to be dealt with. Should a machine initiate further copies before it has received the entire file? (This is tricky because of link or server failures.) How should the knowledge of machines which do not yet have copies of the file be shared? (There can be a central repository or servers can simply check others by random selection.) If the bandwidth between machines in a data center is not a constant, how should the selections be made? (Servers close to each other, e.g., in the same rack, should prefer communicating with each other.)

Finally, it should be mentioned that there are open source solutions to this problem, e.g., Unison and BitTorrent, which would be a good place to start.

20.18 Design the World Wide Web

Design the World Wide Web. Specifically, describe what happens when you enter a URL in a browser address bar, and press return.

Hint: Follow the flow of information.

Solution: At the network level, the browser extracts the domain name component of the URL, and determines the IP address of the server, e.g., through a call to a Domain Name Server (DNS), or a cache lookup. It then communicates using the HTTP protocol with the server. HTTP itself is built on top of TCP/IP, which is responsible for routing, reassembling, and resending packets, as well as controlling the transmission rate.

The server determines what the client is asking for by looking at the portion of the URL that comes after the domain name, and possibly also the body of the HTTP request. The request may be for something as simple a file, which is returned by the webserver; HTTP spells out a format by which the type of the returned file is specified. For example, the URL http://go.com/imgs/abc.png may encode a request for the file whose hierarchical name is imgs/abc.png relative to a base directory specified at configuration to the web server.

The URL may also encode a request to a service provided by the web server. For example, http://go.com/lookup/flight?num=UA37,city=AUS is a request to the lookup/flight service, with an argument consisting of two attribute-value pair. The service could be implemented in many ways, e.g., Java code within the server, or a Common Gateway Interface (CGI) script written in Perl. The service generates a HTTP response, typically HTML, which is then returned to the browser. This response could encode data which is used by scripts running in the browser. Common data formats include JSON and XML.

The browser is responsible for taking the returned HTML and displaying it on the client. The rendering is done in two parts. First, a parse tree (the DOM) is generated from the HTML, and then a rendering library "paints" the screen. The returned HTML may include scripts written in JavaScript. These are executed by the browser, and they can perform actions like making requests and updating the DOM based on the responses—this is how a live stock ticker is implemented. Styling attributes (CSS) are commonly used to customize the look of a page.

Many more issues exist on both the client and server side: security, cookies, HTML form elements, HTML styling, and handlers for multi-media content, to name a few.

20.19 Estimate the hardware cost of a photo sharing app

Estimate the hardware cost of the server hardware needed to build a photo sharing app used by every person on the earth.

Hint: Use variables to denote quantities and costs, and relate them with equations. Then fill in reasonable values.

Solution: The cost is a function of server CPU and RAM resources, storage, and bandwidth, as well as the number and size of the images that are uploaded each day. We will create an estimate based on unit costs for each of these. We assume a distributed architecture in which images are spread ("sharded") across servers.

Assume each user uploads i images each day with an average size of s bytes, and that each image is viewed v times. After d days, the storage requirement is $isdN$, where N is the number of users. Assuming $v \gg 1$, i.e., most images are seen many times, the server cost is dominated by the time to serve the images. The servers are required to serve up Niv images and $Nivs$ bytes each day. Assuming a server can handle h HTTP requests per second and has an outgoing bandwidth of b bytes per second, the number of required servers is $\max(Niv/Th, Nivs/Tb)$, where T is the number of seconds in a day.

Reasonable values for N, i, s, and v are 10^{10}, 10, 10^5, and 100. Reasonable values for h and b are 10^4 and 10^8. There are approximately 10^5 seconds in a day. Therefore the number of servers required is $\max((10^{10} \times 10 \times 100)/(10^5 \times 10^4), (10^{10} \times 10 \times 100 \times 10^5)/(10^5 \times 10^8)) = 10^5$. Each server would cost \$1000, so the total cost would be of the order of 100 million dollars.

Storage costs are approximately \$0.1 per gigabyte, and we add $Nis = 10^{10} \times 10 \times 10^5$ bytes each day, so each day we need to add a million dollars worth of storage.

The above calculation leaves out many costs, such as electricity, cooling, and network. It also neglects computations such as computing trending data and spam analysis. Furthermore, there is no measure of the cost of redundancy, such as replicated storage, or the ability to handle nonuniform loads. Nevertheless, it is a decent starting point. What is remarkable is the fact that the entire world can be connected through images at a very low cost—pennies per person.

CHAPTER

21 Language Questions

The limits of my language means the limits of my world.

— L. WITTGENSTEIN

21.1 Garbage Collection

What is garbage collection? Explain it in the context of Python.

Solution: Garbage collection is the process of finding data objects in a running program that cannot be accessed in the future, and to reclaim the resources, particularly memory, used by those objects. A garbage-collected language is one in which the garbage collection happens automatically—Java, C#, Python, and most scripting languages are examples. C is a notable example of a nongarbage-collected language—the programmer is responsible for knowing when to allocate and deallocate memory.

- Most garbage-collected languages used either reference counting (track the number of references to an object) or tracing (finding objects that are reachable by a sequence of references from certain "root" objects, and considering the rest as "garbage" and collecting them). Python uses reference counting, which has the benefit that it can immediately reclaim objects when the reference count goes to 0; the cost for this is the need for storing an additional integer-value per object. Tracing, which is used in Java, has the benefit that it can be performed in a separate thread, which makes it higher performance. A weakness of tracing is that when the garbage collector runs, it pauses all threads—this leads to nondeterministic performance (sporadic pauses).
- A challenge with reference counting is "reference cycles"—objects A and B reference each other, e.g., $A.u = B$ and $B.v = A$. The reference count never drops below 1, even if nothing else references A and B. The garbage collector periodically looks for these, and removes them.
- Garbage collectors used heuristics for speed. For example, empirically, recently created objects are more likely to be dead. Therefore, as objects are created, they are assigned to generations, and younger generations are examined first.

21.2 Closure

What does the following program print, and why?

```
increment_by_i = [lambda x: x + i for i in range(10)]

print(increment_by_i[3](4))
```

Solution: The program prints 13 (=9 + 4) rather than the 7 (=3 + 4) than might be expected. This is because the the functions created in the loop have the same scope. They use the same variable name, and consequently, all refer to the same variable, *i*, which is 10 at the end of the loop, hence the 13 (=9 + 4).

There are many ways to get the desired behavior. A reasonable approach is to return the lambda from a function, thereby avoiding the naming conflict.

```
def create_increment_function(x):
    return lambda y: y + x


increment_by_i = [create_increment_function(i) for i in range(10)]

print(increment_by_i[3](4))
```

21.3 Shallow and deep copy

Describe the differences between a shallow copy and a deep copy. When is each appropriate, and when is neither appropriate? How is deep copy implemented?

Solution: First, we note that assignment in Python does not copy—it simply a variable to a target. For objects that are mutable, or contain mutable items, a copy is needed if we want to change the copy without changing the original. For example, if `A = [1, 2, 4, 8]` and we assign `A` to `B`, i.e., `B = A`, then if we change `B`, e.g., `B.append(16)`, then `A` will also change.

There are two ways to make a copy: shallow and deep. These are implemented in the copy module, specifically as `copy.copy(x)` and `copy.deepcopy(x)`. A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original. A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

As a concrete example, suppose `A = [[1, 2, 3],[4, 5, 6]]` and `B = copy.copy(A)`. Then if we update `B` as `B[0][0] = 0`, `A[0][0]` also changes to `0`. However, if `B = copy.deepcopy(A)`, then the same update will leave `A` unchanged.

The difference between shallow and deep copying is only relevant for compound objects, that is objects that contain other objects, such as lists or class instances. If `A = [1, 2, 3]` and `B = copy.copy(A)`, then if we update `B` as `B[0] = 0`, `A` is unchanged.

Diving under the hood, deep copy is more challenging to implement since recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) result in infinite recursion in a naive implementation of deep copy. Furthermore, since a deep copy copies everything, it may copy too much, e.g., book-keeping data structures that should be shared even between copies. The `copy.deepcopy()` function avoids these problems by caching objects already copied, and letting user-defined classes override the copying operation. In particular, a class can override one or both of `__copy__()` and `__deepcopy__()`.

Copying is defensive—it may avoided if it's known that the client will not mutate the object. Similarly, if the object is itself immutable, e.g., a tuple, then there's no need to copy it.

What is the difference between an iterator and a generator?

Solution: First, recall what an iterator is—it's any object that has `__iter__()` and `__next__()` methods. The first returns the iterator object itself, and is used is used in `for` and `in` statements. The second method returns the next value in the iteration—if there is no more items it raises the `StopIteration` exception. Here is an iterator that returns numbers that increment by a random amount between 0 and 1, stopping at a limit specified in the constructor.

```
# Iterable object, implements __iter__(), __next__().
class RandomIncrement():
    def __init__(self, limit):
        self._offset = 0.0
        self._limit = limit

    def __iter__(self):
        return self

    def __next__(self):
        self._offset += random.random()
        if (self._offset > self._limit):
            raise StopIteration()
        return self._offset

    # Call this to change the stop condition. It's safe to interleave this with
    # usage of the iterator.
    def increment_limit(self, increment_amount):
        self._limit += increment_amount
```

A generator is an easy way to create iterators. As a concrete example, here is the same iterator expressed as a generator. Note the use of the keyword `yield`.

```
def random_iterator(limit):
    offset = 0
    while True:
        offset += random.random()
        if (offset > limit):
            raise StopIteration()
        yield offset
```

A generator uses the function call stack to implicitly store the state of the iterator—this can simplify the writing of an iterator compared to writing the same iterator as an explicit class; it also helps readability.

Every generator is an iterator, but the converse is not true. In particular, an iterator can be a full-blown class, and can therefore offer additional functionality. For example, it's easy to add a method to the iterator class above to change the iteration limit—this is impossible with the generator.

21.5 @decorator

Explain what a decorator is, with an example that shows why its useful.

Solution: First, recall that functions are first class objects in Python, that is, they can be passed as arguments to other functions, and returned by functions. Additionally, functions can be defined within other functions. In certain contexts, these facts can greatly simplify writing code. For example, here is code that prints the time take to run a function.

```
def time_function(f):
    '''
    Print how long a function takes to compute.
    '''
    begin = time.time()
    result = f()
    end = time.time()
    print("Function call took " + str(end - begin) + " seconds to execute.")
    return result


def foo():
    print("I am foo()")


def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))


time_function(foo)
# functools.partial() take a function and positional arguments to it and
# creates a new function with those arguments preassigned to the specified
# values.
time_function(functools.partial(ackermann, 3, 4))
```

Python provides syntactic sugaring to simplify this process in the form of the `@decorator` construct. The annotation `@time_function` amounts to the following: `foo = @time_function(foo)`.

```
def time_function(f):
    def wrapper(*args, **kwargs):
        begin = time.time()
        result = f(*args, **kwargs)
        end = time.time()
        print("Function call with arguments {all_args} took ".format(
            all_args="\t".join((str(args), str(kwargs)))) + str(end - begin) +
              " seconds to execute.")
        return result

    return wrapper


@time_function
def foo():
    print("I am foo()")
```

```
@time_function
def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))


@time_function
def bar(*args, **kwargs):
    print(sum(args) * sum(kwargs.values()))
```

Other common uses of decorators are checking access control on function backing up REST endpoints, and adding a delay to the end of a function to rate-limit it.

21.6 List vs tuple

In what ways are lists and tuples similar, and in what ways are they different?

Solution: Lists, e.g., `[6, 28, 496, 8128]` and tuples, e.g., `(3.14, "pi", [True, False, True])` are similar in that both represent sequences. Both use the same syntax to access, the i-th element of the sequence, and both support the `in` operator for membership checking.

The key difference between a tuple and a list is that a tuple is immutable—you cannot change the element at index i, or add/delete from a tuple, all of which are possible with a list.

There are numerous benefits of immutability, the most important of which immutable objects are more container-friendly and thread-safe. Elaborating on container-friendliness, if a mutable object is inserted in a set, changed, and then the changed objected is looked up, the lookup will fail (because the hashcode has changed from when it was added to the set). For this reason, tuples are can be put in sets, and used as map keys, but lists cannot.

For the sake of completeness, it's worth pointing out that it's possible to have a mutable object in a tuple, and as a result such a tuple is not longer immutable, as illustrated below:

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

    def __eq__(self, other):
        return isinstance(other, A) and self.x == other.x

    def __hash__(self):
        return self.x * 113 + 119


u = A(42)
v = A(42)
U = (u, )
V = (v, )
```

```
S = set([U])
print(U in S)  # Prints True.
print(V in S)  # Also prints True, since we implemented equals and hash.
u.x = 28
print(U in S)  # Prints False, since u has changed.
```

Some less important differences include the following:

- Conventionally, tuples are used to represent heterogeneous groups of values, where lists are used for homogeneous values, as in the example above.
- Tuples are slightly faster to build and access, and have a smaller memory footprint.

21.7 *ARGS AND *KWARGS

What are *args and *kwargs? Where are they appropriate?

Solution: Both are used to pass a variable number of arguments to a function. The first, *args, is used to pass a variable length argument list, e.g.,

```
def foo(u, v, *args):
    print('u,v = ' + str((u, v)))
    for i in range(len(args)):
        print('args {0} = {1}'.format(str(i), str(args[i])))


foo(1, 'euler', 2.71, [6, 28])
'''
Alternative call.
With the *, foo gets 4 arguments, not 3. Prints
u,v = (1, 'euler')
args 0 = 2.71
args 1 = [6, 28]
'''
args = (2.71, [6, 28])
foo(1, 'euler', *args)
'''
Without the *, foo gets 3 arguments, not 4. Prints
u,v = (1, 'euler')
args 0 = (2.71, [6, 28])
'''
foo(1, 'euler', args)
```

Some points:

- It's not important that the parameter referenced in the function be called args—it could just as well be called A or varargs (though args is idiomatic);
- it is important that the * be specified;
- the * argument must appear after all the regular arguments (if any); and
- * can be used at the call site too.

The second, **kwargs, is used when passing a variable number of keyword arguments to a function.

```
def foo(u, v, *args, **kwargs):
    print('u,v = ' + str((u, v)))
    for i in range(len(args)):
        print('args {0} = {1}'.format(str(i), str(args[i])))
    for (keyword, value) in kwargs.items():
        print('keyword,value =  ' + str((keyword, value)))


foo(1, 'euler', 2.71, [6, 28], name='cfg', rank=1)
# Alternate call
args = (2.71, [6, 28])
kwargs = {'name': 'cfg', 'rank': 1}
foo(1, 'euler', *args, **kwargs)
```

Some points:

- It's not important that the parameter referenced in the function be called `args`—it could just as well be called `D` or `argdict`;
- it is important is the ** be specified;
- the ** argument must appear after all the regular named parameters and the * argument (if any);
- ** can be used at the call site too; and
- keyword arguments are quite different from named arguments, wherein the names of the arguments are specified in the function itself.

21.8 Python code

Rewrite the following program using a Pythonic style.

```
def compute_top_k_variance(students, scores, k):
    '''
    students and scores are equal length lists of strings and floats,
    respectively. The function computes for each string that appears at least
    k times in the list the variance of the top k scores that correspond to it.
    Strings that appear fewer than k times are not considered.
    '''
    counts = {}
    for i in range(len(students)):
        if students[i] not in counts:
            counts[students[i]] = 1
        else:
            counts[students[i]] += 1

    all_scores = {}
    for key in counts:
        if counts[key] >= k:
            all_scores[key] = []

    for i in range(len(students)):
        if students[i] in all_scores:
            all_scores[students[i]].append(scores[i])
```

```
    top_k_scores = {}
    for key in all_scores:
        sorted_scores = sorted(all_scores[key])
        top_k_scores[key] = []
        for i in range(k):
            top_k_scores[key].append(sorted_scores[len(sorted_scores) - 1 - i])

    result = {}
    for key in top_k_scores:
        total = 0
        for score in top_k_scores[key]:
            total += score
        mean = total / k
        variance = 0
        for score in top_k_scores[key]:
            variance = variance + (score - mean) * (score - mean)
        result[key] = variance

    return result
```

Solution: The functions `zip()`, `functools.reduce()`, `heapq.nlargest()`, and `sum()`, directly replace boilerplate loops. The `collections.defaultdict` container and dictionary comprehension remove the need for explicit initialization.

```
def compute_top_k_variance(students, scores, k):
    all_scores = collections.defaultdict(list)
    for student, score in zip(students, scores):
        all_scores[student].append(score)

    top_k_scores = {
        student: heapq.nlargest(k, scores)
        for student, scores in all_scores.items() if len(scores) >= k
    }

    return {
        student: functools.reduce(
            lambda variance, score: variance + (score - mean)**2, scores, 0)
        for student, scores, mean in (
            (student, scores, sum(scores) / k)
            for student, scores in top_k_scores.items())
    }
```

21.9 Exception Handling

Briefly describe exception handling in Python, paying special attention to the roles played by `try`, `except`, `else`, `finally`, and `raise`. Rewrite the following program using exceptions to make it more robust.

```
def get_col_sum(filename, col):
    # May raise IOError.
    csv_file = open(filename)
    csv_reader = csv.reader(csv_file)
```

```
    running_sum = 0
    for row in csv_reader:
        value = row[col]
        running_sum += int(value)
    csv_file.close()
    print("Sum = " + str(running_sum))
```

Solution: A statement or expression it may result in an error when an attempt is made to execute it. Such errors are called exceptions. They do not always entail exiting the program. For example, if a user enters a filename, and no corresponding file is found, or the user does not have read permissions, the user can be prompted again. Such checks can be made using a `try` block.

The `try` block can be followed by an `except` block, a `finally` block, or an `except` block followed by a `finally` block or an `except` block followed by an `else` block followed by a `finally` block.

If a `finally` block exists, it is always executed, regardless of whether an exception was raised and/or caught. Here is an example. Note that there is no `except` block—any exception raised in the `try` block is propagated up.

```
def get_value(filename, key):
    handle = open(filename)
    try:
        file_contents = handle.read()
        js_text = json.loads(file_contents)
        return js_text[key]
    finally:
        # Prevent resource leak if there is an error parsing file_contents.
        handle.close()
```

The code below shows basic `try/except` usage. Note the duplicated code used to avoid the resource leak.

```
def get_value(filename, key):
    handle = open(filename)
    try:
        file_contents = handle.read()
        js_text = json.loads(file_contents)
        handle.close()
        return (True, js_text[key])
    except ValueError:
        handle.close()
        return (False, )
```

The duplicated code can be avoided by using `try/except/finally`.

```
def get_value(filename, key):
    handle = open(filename)
    try:
        file_contents = handle.read()
        js_text = json.loads(file_contents)
        return (True, js_text[key])
    except ValueError:
        return (False, )
    finally:
        handle.close()
```

Here's the original program rewritten using `try/except/else/finally` blocks to make it more robust. In particular, the `else` shows what code is executed in the absence of exceptions. Note how we use `raise` to create/propagate exceptions upwards.

```
class ColSumCsvParseException(Exception):
    def __init__(self, *args):
        Exception.__init__(self, *args)
        self.line_number = args[1]


def get_col_sum(filename, col):
    # may raise IOError, will propagate to caller
    csv_file = open(filename)
    csv_reader = csv.reader(csv_file)
    running_sum = line_number = 0
    try:
        for row in csv_reader:
            if col >= len(row):
                raise IndexError("Not enough entries in row " + str(row))
            value = row[col]
            # We will skip rows for which the corresponding columns cannot be
            # parsed to an int, logging the fact.
            try:
                running_sum += int(value)
            except ValueError:
                print("Cannot convert " + value + " to int, ignoring")
            line_number += 1
    except csv.Error:
        # Programs should raise exceptions appropriate to their level of
        # abstraction, so we propagate the csv.Error upwards as a
        # ColSumCsvParseException.
        print("In csv.Error handler")
        raise ColSumCsvParseException("Error processing csv", line_number)
    else:
        print("Sum = " + str(running_sum))
    finally:
        # Ensure there is no resource leak.
        csv_file.close()
        return running_sum
```

There are a number of built-in exception types in Python, e.g., `IndexError`, `FloatingPointError`, `EnvironmentError`, `ValueError`, etc., and these should be used whenever possible.

21.10 Scoping

Explain the rules for variable scope.

Solution: There are two (nonexclusive) possibilities: the variable appears in an expression, and the variable is being assigned to.

When the variable appears in an expression, Python searches for it in the following order:

(1.) The current function.

(2.) Enclosing scopes, e.g., containing functions.

(3.) The module containing the code (also referred to as the global scope)

(4.) The built-in scope, e.g., `open`

(A `NameError` is raised if none of these contain a defined variable with the given name.)

In the simplest case, the variable is defined already in the current scope, in which case the assignment happens to it. Otherwise, if the assignment happens outside a function, it defines a new global variable; if it happens within a function, it defines a new variable whose scope is the function that contains the assignment.

To make an assignment to a variable not defined in the current scope within a function happen to a global variable, define a new variable in the current scope, that variable should be declared as `global` in the function. In Python3, to have an assignment to a variable in nested function use an enclosing function's scope, use `nonlocal`: this will make the program search outwards through the enclosing scopes (but not to the module level) for that variable.

The scoping rules seek to prevent variables local to a function from polluting the containing functions and module. They can lead to scoping bugs though, so as a rule names should be kept distinct.

Here is a program that illustrates these rules.

```
x, y, z = 'global-x', 'global-y', 'global-z'


def basic_scoping():
    print(x)  # global-x
    y = 'local-y'
    global z
    z = 'local-z'


basic_scoping()
print(x, y, z)  # global-x global-y local-z


def inner_outer_scoping():
    def inner1():
        print(x)  # outer-x

    def inner2():
        x = 'inner2-x'
        print(x)  # inner2-x

    def inner3():
        nonlocal x
        x = 'inner3-x'
        print(x)  # inner3-x

    x = "outer-x"
    inner1(), inner2(), inner3()
    print(x)  # inner3-x


inner_outer_scoping()
print(x, y, z)  # global-x global-y local-z
```

```
def outer_scope_error():
    def inner():
        try:
            x = x + 321
        except NameError:
            print('Error: x is local, and so x + 1 is not defined yet')

    x = 123
    inner()


outer_scope_error()  # prints 'Error: ...'


def outer_scope_array_no_error():
    def inner():
        x[0] = -x[0]  # x[0] isn't a variable, it's resolved from outer x.

    x = [314]
    inner()
    print(x[0])  # -314


outer_scope_array_no_error()
```

21.11 Function arguments

What are positional, keyword, and default arguments to a function. What does the following program output, and why?

```
def foo(x=[]):
    x.append(1)
    return x


res = foo()
print(res)
res = foo()
print(res)
```

Solution: On the calling side, some function arguments can be specified by name. These arguments have come after all of the unnamed arguments (also known as positional arguments). Consider the following function.

```
def foo(x, y, z):
    return x * x + y * y + z * z
```

It can be called using positional arguments, or keyword arguments as follows.

```
foo(1, 2, 3)
```

```
foo(x=1, y=2, z=3)
foo(1, y=2, z=3)
foo(1, z=3, y=3)
foo(z=3, y=3, x=1)
# This is a syntax error, keyword arguments must follow positional arguments.
foo(x=1, 2, z=3)
# This is a syntax error, a variable cannot be assigned twice.
foo(1, x=2, z=3, y=4)
```

Keywords arguments have the following benefits.

- The function call is much clearer—it's harder to mistakenly exchange two values.
- They make it easier to refactor a function into having more arguments. Specifically, the combination of keyword arguments and default values means that old code does not have to be touched even when new arguments are added.

When a function is defined, its arguments can be specified to have default values; if the function is called without the argument, the argument gets its default value. There must be default values for all arguments which are not specified in calls to the function.

One subtlety with default arguments is that they are evaluated once, specifically when the module is loaded, and are shared across all callers. For this reason, the program given in the example prints `[1,1]` after the second call—the first call updates the default argument to `[1]`. As a rule, mutable arguments should have `None` as their default values. Then the function can test the argument in its body—if it is `None`, the function can assign it to the desired default value. See the program below for an example.

```
def read_file_as_array_buggy(filename, default=[]):
    try:
        filehandle = open(filename)
        return filename.read().split()
    except Exception:
        return default


def read_file_as_array_works(filename, default=None):
    if default is None:
        default = []
    try:
        filehandle = open(filename)
        return filename.read().split()
    except Exception:
        return default


A = read_file_as_array_buggy("does_not_exist.txt")
A.append("first")
B = read_file_as_array_buggy("does_not_exist.txt")
B.append("second")
print(B)  # ['first', 'second']
A = read_file_as_array_works("does_not_exist.txt")
A.append("first")
B = read_file_as_array_works("does_not_exist.txt")
B.append("second")
```

```
print(B)  # ['second']
```

CHAPTER

22 Object-Oriented Design

One thing expert designers know not to do is solve every problem from first principles.

— *"Design Patterns: Elements of Reusable Object-Oriented Software,"* E. GAMMA, R. HELM, R. E. JOHNSON, AND J. M. VLISSIDES, 1994

A class is an encapsulation of data and methods that operate on that data. Classes match the way we think about computation. They provide encapsulation, which reduces the conceptual burden of writing code, and enable code reuse, through the use of inheritance and polymorphism. However, naive use of object-oriented constructs can result in code that is hard to maintain.

A design pattern is a general repeatable solution to a commonly occurring problem. It is not a complete design that can be coded up directly—rather, it is a description of how to solve a problem that arises in many different situations. In the context of object-oriented programming, design patterns address both reuse and maintainability. In essence, design patterns make some parts of a system vary independently from the other parts.

Adnan's Design Pattern course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

22.1 TEMPLATE METHOD VS. STRATEGY

Explain the difference between the template method pattern and the strategy pattern with a concrete example.

Solution: Both the template method and strategy patterns are similar in that both are behavioral patterns, both are used to make algorithms reusable, and both are general and very widely used. However, they differ in the following key way:

- In the template method, a skeleton algorithm is provided in a superclass. Subclasses can override methods to specialize the algorithm.
- The strategy pattern is typically applied when a family of algorithms implements a common interface. These algorithms can then be selected by clients.

As a concrete example, consider a sorting algorithm like quicksort. Two of the key steps in quicksort are pivot selection and partitioning. Quicksort is a good example of a template method—subclasses can implement their own pivot selection algorithm, e.g., using randomized median finding or selecting an element at random, and their own partitioning method, e.g., using the DNF partitioning algorithms in Solution 5.1 on Page 40.

Since there may be multiple ways in which to sort elements, e.g., student objects may be compared by GPA, major, name, and combinations thereof, it's natural to make the comparison operation used by the sorting algorithm an argument to quicksort. One way to do this is to pass

quicksort an object that implements a compare method. These objects constitute an example of the strategy pattern, as do the objects implementing pivot selection and partitioning.

There are some other smaller differences between the two patterns. For example, in the template method pattern, the superclass algorithm may have "hooks"—calls to placeholder methods that can be overridden by subclasses to provide additional functionality. Sometimes a hook is not implemented, thereby forcing the subclasses to implement that functionality; some times it offers a "no-operation" or some baseline functionality. There is no analog to a hook in a strategy pattern.

Note that there's no relationship between the template method pattern and template meta-programming (a form of generic programming favored in C++).

22.2 Observer pattern

Explain the observer pattern with an example.

Solution: The observer pattern defines a one-to-many dependency between objects so that when one object changes state all its dependents are notified and updated automatically.

The observed object must implement the following methods.

- Register an observer.
- Remove an observer.
- Notify all currently registered observers.

The observer object must implement the following method.

- Update the observer. (Update is sometimes referred to as notify.)

As a concrete example, consider a service that logs user requests, and keeps track of the 10 most visited pages. There may be multiple client applications that use this information, e.g., a leaderboard display, ad placement algorithms, recommendation system, etc. Instead of having the clients poll the service, the service, which is the observed object, provides clients with register and remove capabilities. As soon as its state changes, the service enumerates through registered observers, calling each observer's update method.

Though most readily understood in the context of a single program, where the observed and observer are objects, the observer pattern is also applicable to distributed computing.

22.3 Push vs. pull observer pattern

In the observer pattern, subjects push information to their observers. There is another way to update data—the observers may "pull" the information they need from the subject. Compare and contrast these two approaches.

Solution: Both push and pull observer designs are valid and have tradeoffs depending on the needs of the project. With the push design, the subject notifies the observer that its data is ready and includes the relevant information that the observer is subscribing to, whereas with the pull design, it is the observer's job to retrieve that information from the subject.

The pull design places a heavier load on the observers, but it also allows the observer to query the subject only as often as is needed. One important consideration is that by the time the observer retrieves the information from the subject, the data could have changed. This could be a positive or negative result depending on the application. The pull design places less responsibility on the subject for tracking exactly which information the observer needs, as long as the subject knows

when to notify the observer. This design also requires that the subject make its data publicly accessible by the observers. This design would likely work better when the observers are running with varied frequency and it suits them best to get the data they need on demand.

The push design leaves all of the information transfer in the subject's control. The subject calls update for each observer and passes the relevant information along with this call. This design seems more object-oriented, because the subject is pushing its own data out, rather than making its data accessible for the observers to pull. It is also somewhat simpler and safer in that the subject always knows when the data is being pushed out to observers, so you don't have to worry about an observer pulling data in the middle of an update to the data, which would require synchronization.

22.4 Singletons and Flyweights

Explain the differences between the singleton pattern and the flyweight pattern. Use concrete examples.

Solution: The singleton pattern ensures a class has only one instance, and provides a global point of access to it. The flyweight pattern minimizes memory use by sharing as much data as possible with other similar objects. It is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

A common example of a singleton is a logger. There may be many clients who want to listen to the logged data (console, file, messaging service, etc.), so all code should log to a single place.

A common example of a flyweight is string interning—a method of storing only one copy of each distinct string value. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are usually stored in a hash table. Since multiple clients may refer to the same flyweight object, for safety flyweights should be immutable.

There is a superficial similarity between singleton and flyweight: both keep a single copy of an object. There are several key differences between the two:

- Flyweights are used to save memory. Singletons are used to ensure all clients see the same object.
- A singleton is used where there is a single shared object, e.g., a database connection, server configurations, a logger, etc. A flyweight is used where there is a family of shared objects, e.g., objects describing character fonts, or nodes shared across multiple binary search trees.
- Flyweight objects are invariable immutable. Singleton objects are usually not immutable, e.g., requests can be added to the database connection object.
- The singleton pattern is a creational pattern, whereas the flyweight is a structural pattern.

In summary, a singleton is like a global variable, whereas a flyweight is like a pointer to a canonical representation.

Sometimes, but not always, a singleton object is used to create flyweights—clients ask the singleton for an object with specified fields, and the singleton checks its internal pool of flyweights to see if one exists. If such an object already exists, it returns that, otherwise it creates a new flyweight, add it to its pool, and then returns it. (In essence the singleton serves as a gateway to a static factory.)

What is the difference between a class adapter and an object adapter?

Solution: The adapter pattern allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code.

There are two ways to build an adapter: via subclassing (the class adapter pattern) and composition (the object adapter pattern). In the class adapter pattern, the adapter inherits both the interface that is expected and the interface that is pre-existing. In the object adapter pattern, the adapter contains an instance of the class it wraps and the adapter makes calls to the instance of the wrapped object.

Here are some remarks on the class adapter pattern.

- The class adapter pattern allows re-use of implementation code in both the target and adaptee. This is an advantage in that the adapter doesn't have to contain boilerplate pass-throughs or cut-and-paste reimplementation of code in either the target or the adaptee.
- The class adapter pattern has the disadvantages of inheritance (changes in base class may cause unforeseen misbehaviors in derived classes, etc.). The disadvantages of inheritance are made worse by the use of two base classes, which also precludes its use in languages like Java (prior to Java 1.8) that do not support multiple inheritance.
- The class adapter can be used in place of either the target or the adaptee. This can be advantage if there is a need for a two-way adapter. The ability to substitute the adapter for adaptee can be a disadvantage otherwise as it dilutes the purpose of the adapter and may lead to incorrect behavior if the adapter is used in an unexpected manner.
- The class adapter allows details of the behavior of the adaptee to be changed by overriding the adaptee's methods. Class adapters, as members of the class hierarchy, are tied to specific adaptee and target concrete classes.

As a concrete example of an object adapter, suppose we have legacy code that returns objects of type stack. Newer code expects inputs of type deque, which is more general than stack (but does not subclass stack). We could create a new type, stack-adapter, which implements the deque methods, and can be used anywhere deque is required. The stack-adapter class has a field of type stack—this is referred to as object composition. It implements the deque methods with code that uses methods on the composed stack object. Deque methods that are not supported by the underlying stack throw unsupported operation exceptions. In this scenario, the stack-adapter is an example of an object adapter.

Here are some comments on the object adapter pattern.

- The object adapter pattern is "purer" in its approach to the purpose of making the adaptee behave like the target. By implementing the interface of the target only, the object adapter is only useful as a target.
- Use of an interface for the target allows the adaptee to be used in place of any prospective target that is referenced by clients using that interface.
- Use of composition for the adaptee similarly allows flexibility in the choice of the concrete classes. If adaptee is a concrete class, any subclass of adaptee will work equally well within the object adapter pattern. If adaptee is an interface, any concrete class implementing that interface will work.

- A disadvantage is that if target is not based on an interface, target and all its clients may need to change to allow the object adapter to be substituted.

Variant: The UML diagrams for decorator, adapter, and proxy look identical, so why is each considered a separate pattern?

22.6 Creational Patterns

Explain what each of these creational patterns is: builder, static factory, factory method, and abstract factory.

Solution: The idea behind the builder pattern is to build a complex object in phases. It avoids mutability and inconsistent state by using an mutable inner class that has a build method that returns the desired object. Its key benefits are that it breaks down the construction process, and can give names to steps. Compared to a constructor, it deals far better with optional parameters and when the parameter list is very long.

A static factory is a function for construction of objects. Its key benefits are as follow: the function's name can make what it's doing much clearer compared to a call to a constructor. The function is not obliged to create a new object—in particular, it can return a flyweight. It can also return a subtype that's more optimized, e.g., it can choose to construct an object that uses an integer in place of a Boolean array if the array size is not more than the integer word size.

A factory method defines interface for creating an object, but lets subclasses decide which class to instantiate. The classic example is a maze game with two modes—one with regular rooms, and one with magic rooms. The program below uses a template method, as described in Problem 22.1 on Page 333, to combine the logic common to the two versions of the game.

```
from abc import ABC, abstractmethod


class Room(ABC):
    @abstractmethod
    def connect(self, room2):
        pass


class MazeGame(ABC):
    @abstractmethod
    def make_room(self):
        print("abstract make_room")
        pass

    def addRoom(self, room):
        print("adding room")

    def __init__(self):
        room1 = self.make_room()
        room2 = self.make_room()
        room1.connect(room2)
        self.addRoom(room1)
        self.addRoom(room2)
```

This snippet implements the regular rooms. This snippet implements the magic rooms.

```
class MagicMazeGame(MazeGame):
    def make_room(self):
        return MagicRoom()


class MagicRoom(Room):
    def connect(self, room2):
        print("Connecting magic room")
```

Here's how you use the factory to create regular and magic games.

```
ordinary_maze_game = ordinary_maze_game.OrdinaryMazeGame()
magic_maze_game = magic_maze_game.MagicMazeGame()
```

A drawback of the factory method pattern is that it makes subclassing challenging.

An abstract factory provides an interface for creating families of related objects without specifying their concrete classes. For example, a class `DocumentCreator` could provide interfaces to create a number of products, such as `createLetter()` and `createResume()`. Concrete implementations of this class could choose to implement these products in different ways, e.g., with modern or classic fonts, right-flush or right-ragged layout, etc. Client code gets a `DocumentCreator` object and calls its factory methods. Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. The price for this flexibility is more planning and upfront coding, as well as and code that may be harder to understand, because of the added indirections.

22.7 Libraries and design patterns

Why is there no library of design patterns so a developers do not have to write code every time they want to use them?

Solution: There are several reasons that design patterns cannot be delivered as a set of libraries. The key idea is that patterns cannot be cleanly abstracted from the objects and the processes they are applicable to. More specifically, libraries provide the implementations of algorithms. In contrast, design patterns provide a higher level understanding of how to structure classes and objects to solve specific types of problems. Another difference is that it's often necessary to use combinations of different patterns to solve a problem, e.g., Model-View-Controller (MVC), which is commonly used in UI design, incorporates the Observer, Strategy, and Composite patterns. It's not reasonable to come up with libraries for every possible case.

Of course, many libraries take advantage of design patterns in their implementations: sorting and searching algorithms use the template method pattern, custom comparison functions illustrate the strategy pattern, string interning is an example of the flyweight pattern, typed-I/O shows off the decorator pattern, etc.

Variant: Give examples of commonly used library code that use the following patterns: template, strategy, observer, singleton, flyweight, static factory, decorator, abstract factory.

Variant: Compare and contrast the iterator and composite patterns.

CHAPTER

23 Common Tools

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages.

— *"The UNIX TimeSharing System,"*
D. RITCHIE AND K. THOMPSON, 1974

The problems in this chapter are concerned with tools: version control systems, scripting languages, build systems, databases, and the networking stack. Such problems are not commonly asked—expect them only if you are interviewing for a specialized role, or if you claim specialized knowledge, e.g., network security or databases. We emphasize these are vast subjects, e.g., networking is taught as a sequence of courses in a university curriculum. Our goal here is to give you a flavor of what you might encounter. Adnan's Advanced Programming Tools course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

Version control

Version control systems are a cornerstone of software development—every developer should understand how to use them in an optimal way.

23.1 MERGING IN A VERSION CONTROL SYSTEM

What is merging in a version control system? Specifically, describe the limitations of line-based merging and ways in which they can be overcome.

Solution:

Modern version control systems allow developers to work concurrently on a personal copy of the entire codebase. This allows software developers to work independently. The price for this merging: periodically, the personal copies need to be integrated to create a new shared version. There is the possibility that parallel changes are conflicting, and these must be resolved during the merge.

By far, the most common approach to merging is text-based. Text-based merge tools view software as text. Line-based merging is a kind of text-based merging, specifically one in which each line is treated as an indivisible unit. The merging is "three-way": the lowest common ancestor in the revision history is used in conjunction with the two versions. With line-based merging of text files, common text lines can be detected in parallel modifications, as well as text lines that have been

inserted, deleted, modified, or moved. Figure 23.1 on the facing page shows an example of such a line-based merge.

One issue with line-based merging is that it cannot handle two parallel modifications to the same line: the two modifications cannot be combined, only one of the two modifications must be selected. A bigger issue is that a line-based merge may succeed, but the resulting program is broken because of syntactic or semantic conflicts. In the scenario in Figure 23.1 on the next page the changes made by the two developers are made to independent lines, so the line-based merge shows no conflicts. However, the program will not compile because a function is called incorrectly.

In spite of this, line-based merging is widely used because of its efficiency, scalability,and accuracy. A three-way, line-based merge tool in practice will merge 90% of the changed files without any issues. The challenge is to automate the remaining 10% of the situations that cannot be merged automatically.

Text-based merging fails because it does not consider any syntactic or semantic information.

Syntactic merging takes the syntax of the programming language into account. Text-based merge often yields unimportant conflicts such as a code comment that has been changed by different developers or conflicts caused by code reformatting. A syntactic merge can ignore all these: it displays a merge conflict when the merged result is not syntactically correct.

We illustrate syntactic merging with a small example:

```
if (n%2 == 0) then m = n/2;
```

Two developers change this code in a different ways, but with the same overall effect. The first updates it to

```
if (n%2 == 0) then m = n/2 else m = (n-1)/2;
```

and the second developer's update is

```
m = n/2;
```

Both changes to the same thing. However, a textual-merge will likely result in

```
m := n div 2 else m := (n-1)/2
```

which is syntactically incorrect. Syntactic merge identifies the conflict; it is the integrator's responsibility to manually resolve it, e.g., by removing the `else` portion.

Syntactic merge tools are unable to detect some frequently occurring conflicts. For example, in the merge of Versions *1a* and *1b* in Figure 23.1 on the facing page will not compile, since the call to `sum(10)` has the wrong signature. Syntactic merge will not detect this conflict since the program is still syntactically correct. The conflict is a semantic conflict, specifically, a static semantic conflict, since it is detectable at compile-time. (The compile will return something like "function argument mismatch error".)

Technically, syntactic merge algorithms operate on the parse trees of the programs to be merged. More powerful static semantic merge algorithms have been proposed that operate on a graph representation of the programs, wherein definitions and usage are linked, making it easier to identify mismatched usage.

Static semantic merging also has shortcomings. For example, suppose `Point` is a class representing 2*D*-points using Cartesian coordinates, and that this class has a distance function that returns $\sqrt{x^2 + y^2}$. Suppose Alice checks out the project, and subclasses `Point` to create a class that supports polar coordinates, and uses `Points`'s distance function to return the radius. Concurrently, Bob checks out the project and changes `Point`'s distance function to return $|x| + |y|$. Static semantic

merging reports no conflicts: the merged program compiles without any trouble. However the behavior is not what was expected.

Both syntactic merging and semantic merging greatly increase runtimes, and are very limiting since they are tightly coupled to specific programming languages. In practice, line-based merging is used in conjunction with a small set of unit tests (a "smoke suite") invoked with a pre-commit hook script. Compilation finds syntax and static semantics errors, and the tests (hopefully) identify deeper semantic issues.

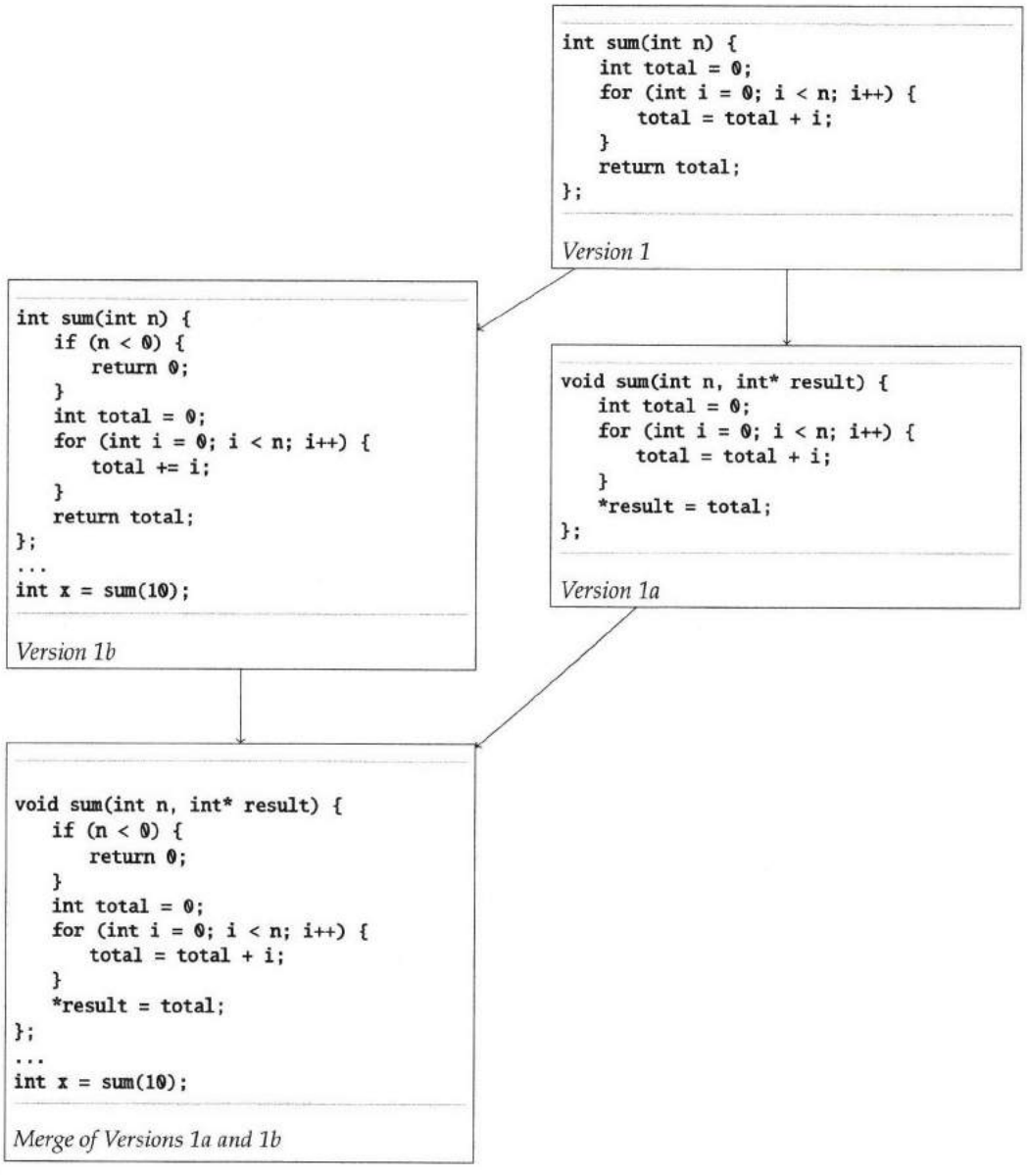


Figure 23.1: An example of a 3-way line based merge.

23.2 Hooks

What are hooks in a version control system? Describe their applications.

Solution: Version control systems such as git and SVN allow users to define actions to take around events such as commits, locking and unlocking files, and altering revision properties. These actions are specified as executable programs known as hook scripts. When the version control system gets to these events, it will check for the existence of a corresponding hook script, and execute it if it exists.

Hook scripts have access to the action as it is taking place, and are passed corresponding command-line arguments. For example, the pre-commit hook is passed the repository path and the transaction ID for the currently executing commit. If a hook script returns a nonzero exit code, the version control system aborts the action, returning the script's standard error output to the user.

The following hook scripts are used most often:

- *pre-commit:* Executed before a change is committed to the repository. Often used to check log messages, to format files, run a test suite, and to perform custom security or policy checking.
- *post-commit:* Executed once the commit has completed. Often used to inform users about a completed commit, for example by sending an email to the team, or update the bug tracking system.

As a rule, hooks should never alter the content of the transaction. At the very least, it can surprise a new developer; in the worst-case, the change can result in buggy code.

Scripting languages

Scripting languages, such as AWK, Perl, and Python were originally developed as tools for quick hacks, rapid prototyping, and gluing together other programs. They have evolved into mainstream programming languages. Some of their characteristics are as follows.

- Text strings are the basic (sometimes only) data type.
- Associative arrays are a basic aggregate type.
- Regexps are usually built in.
- Programs are interpreted rather than compiled.
- Declarations are often not required.

They are easy to get started with, and can often do a great deal with very little code.

23.3 Is scripting more efficient?

Your manager read a study in which a Python program was found to have taken one-tenth the development time then a program written in C++ for the same application, and now demands that you immediately program exclusively in Python. Tell him about five dangers of doing this.

Solution: First, any case study is entirely anecdotal. It's affected by the proficiency of the developers, the nature of the program being written, etc. So no case study alone is sufficient as the basis for a language choice.

Second, switching development environments for any team is an immediate slowdown, since the developers must learn all the new tricks and unlearn all the old ones. If a team wants to make this kind of switch, it needs to have a very long runway.

Third, scripting languages are simply not good for some tasks. For example, many high-performance computing or systems tasks should not be done in Python. The lack of a type system means some serious bugs can go undetected.

Fourth, choosing to program entirely in any language is a risky endeavor. Any decent engineering department should be willing to adopt the best tool for the job at hand.

Finally, Python itself may be a risky choice at this point in time, since there is an ongoing shift from the 2.x branch to 3.x. If you choose to work in 2.x, you likely have a massive refactor into 3.x

coming in the next few years; and if you choose to work in 3.x, you likely will run into libraries you simply cannot use because they don't yet support the latest versions of Python.

23.4 Polymorphism with a scripting language

Briefly, a Java interface is a kind of type specification. To be precise, it specifies a group of related methods with empty bodies. (All methods are nonstatic and public.) Interfaces have many benefits over, for example, abstract classes—in particular, they are a good way to implement polymorphic functions in Java.

Describe the advantages and disadvantages of Python and Java for polymorphism. In particular, how would you implement a polymorphic function in Python?

Solution: Java's polymorphism is first-class. The compiler is aware at compile-time of whatever interfaces (in the general sense) an object exposes—regardless of whether they derive from actual interfaces or from superclasses. As a result, a polymorphic function in Java has a guarantee that any argument to it implements the methods it will call during its execution.

In Python, however, no such safety exists. So there are two tricks to use when implementing a polymorphic function in Python. First, you can specify your classes using multiple inheritance. In this case, many of the superclasses are simply interface specifications. These classes, mix-ins, may implement the desired methods or leave them for the subclass to implement. In effect, they provide a poor-man's interface. (However, they're not as robust as Java's solution since the method resolution order gets tricky in multiple-inheritance scenarios.)

But there is a trickier problem here, which is that even if a class is implemented with a particular method, it's trivial at runtime to overwrite that method, e.g., with `None`, before passing it as an argument. So you still don't have any guarantees. This motivates the second trick, which is the use of "duck typing"—that is, rather than trying to get some safety guarantee that cannot exist in Python, just inspect the object at runtime to see if it exposes the right interface. (The phrase comes from this adage: "If it looks like a duck, quacks like a duck, and acts like a duck, then it's probably a duck".) So, where in Java a developer might write a function that only accepts arguments that implement a `Drawable` interface, in Python you would simply call that object's draw method and hope for the best. Type checking still exists here; it's just deferred to runtime, which implies it may throw exceptions. The function is still quite polymorphic—more so, in some sense, since it can accept literally any object that has a draw method.

Build systems

A program typically cannot be executed immediately after change: intermediate steps are required to build the program and test it before deploying it. Other components can also be affected by changes, such as documentation generated from program text. Build systems automate these steps.

23.5 Dependency analysis

Argue that a build system such as Make which looks purely at the time stamps of source code and derived products when determining when a product needs to be rebuilt can end up spending more time building the product that is needed. How can the you avoid this? Are there any pitfalls to your approach?

Solution: Suppose a source file is changed without its semantics being affected, e.g., a comment is added or it's reformatted. A build system like Make will propagate this change through dependencies, even though subsequent products are unchanged.

One solution to this is as follows. During the reconstruction of a component A, a component called A_{new} is created and compared with the existing A. If A_{new} and A are different, A_{new} is copied to A. Otherwise, A remains unchanged, include the date that it was changed.

This may not always be the reasonable thing to do. For example, even if the source files remain unchanged, environmental differences or changes to the compilation process may lead to the creation of different intermediate targets, which would necessitate a full rebuild. Therefore each intermediate target must be rebuilt in order to compare against the existing components, but it's indeterminate before construction whether the new objects will be used or not.

23.6 ANT vs. Maven

Contrast ANT and Maven

Solution: Historically, ANT was developed in the early days of Java to overcome the limitations of Make. It has tasks as a "primitive"—in Make these must be emulated using pseudo-targets, variables, etc. ANT is cross-platform—since Make largely scripts what you would do at the command-line, a Makefile written for Unix-like platforms will not work out-of-the-box on, say, Windows.

ANT itself has a number of shortcomings. These are overcome in Maven, as described below.

- Maven is declarative whereas Ant is imperative in nature. You tell Maven what are the dependencies of your project—Maven will fetch those dependencies and build the artifact for you, e.g., a war file. In Ant, you have to provide all the dependencies of your project and tell Ant where to copy those dependencies.
- With Maven, you can use "archetypes" to set up your project very quickly. For example you can quickly set up a Java Enterprise Edition project and get to coding in a matter of minutes. Maven knows what should be the project structure for this type of project and it will set it up for you.
- Maven is excellent at dependency management. Maven automatically retrieves dependencies for your project. It will also retrieve any dependencies of dependencies. You never have to worry about providing the dependent libraries. Just declare what library and version you want and Maven will get it for you.
- Maven has the concept of snapshot builds and release builds. During active development your project is built as a snapshot build. When you are ready to deploy the project, Maven will build a release version for you which can be put into a company wide Nexus repository. This way, all your release builds are in one repository that are accessible throughout the company.
- Maven favors convention over configuration. In Ant, you have to declare a bunch of properties before you can get to work. Maven uses sensible defaults and if you follow the Maven recommended project layout, your build file is very small and everything works out of the box.
- Maven has very good support for running automated tests. In fact, Maven will run your tests as part of the build process by default. You do not have to do anything special.
- Maven provides hooks at different phases of the build and you can perform additional tasks as per your needs

Database

Most software systems today interact with databases, and it's important for developers to have basic knowledge of databases.

23.7 SQL vs. NoSQL

Contrast SQL and NoSQL databases.

Solution: A relational database is a set of tables (also known as relations). Each table consists of rows. A row is a set of columns (also known as fields or attributes), which could be of various types (integer, float, date, fixed-size string, variable-size string, etc.). SQL is the language used to create and manipulate such databases. MySQL is a popular relational database.

A NoSQL database provides a mechanism for storage and retrieval of data which is modeled by means other than the tabular relations used in relational databases. MongoDB is a popular NoSQL database. The analog of a table in MongoDB is a collection, which consists of documents. Collections do not enforce a schema. Documents can be viewed as hash maps from fields to values. Documents within a collection can have different fields.

A key benefit of NoSQL databases is simplicity of design: a field can trivially be added to new documents in a collection without this affecting documents already in the database. This makes NoSQL a popular choice for startups which have an agile development environment.

The data structures used by NoSQL databases, e.g., key-value pairs in MongoDB, are different from those used by default in relational databases, making some operations faster in NoSQL. NoSQL databases are typically much simpler to scale horizontally, i.e., to spread documents from a collection across a cluster of machines.

A key benefit of relational databases include support for ACID transactions. ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. For example, there is no way in MongoDB to remove an item from inventory and add it to a customer's order as an atomic operation.

Another benefit of relational database is that their query language, SQL, is declarative, and relatively simple. In contrast, complex queries in a NoSQL database have to be implemented programmatically. (It's often the case that the NoSQL database will have some support for translating SQL into the equivalent NoSQL program.)

23.8 Normalization

What is database normalization? What are its advantages and disadvantages?

Solution: Database normalization is the process of organizing the columns and tables of a relational database to minimize data redundancy. Specifically, normalization involves decomposing a table into less redundant tables without losing information, thereby enforcing integrity and saving space.

The central idea is the use of "foreign keys". A foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table. In simpler words, the foreign key is defined in a second table, but it refers to the unique key in the first table. For example, a table called Employee may use a unique key called employee_id. Another table called Employee Details has a foreign key which references employee_id in order to uniquely identify the relationship between both the tables. The objective is to isolate data so that additions, deletions, and modifications of an attribute can be made in just one table and then propagated through the rest of the database using the defined foreign keys.

The primary drawback of normalization is performance—often, a large number of joins are required to recover the records the application needs to function.

Write SQL commands to create a table appropriate for representing students as well as SQL commands that illustrate how to add, delete, and update students, as well as search for students by GPA with results sorted by GPA.

Then add a table of faculty advisors, and describe how to model adding an advisor to each student. Write a SQL query that returns the students who have an advisor with a specific name.

Solution: Natural fields for a student are name, birthday, GPA, and graduating year. In addition, we would like a unique key to serve as an identifier. A SQL statement to create a student table would look like the following: `CREATE TABLE students (id INT AUTO_INCREMENT, name VARCHAR(30), birthday DATE, gpa FLOAT, graduate_year INT, PRIMARY KEY(id));`. Here are examples of SQL statements for updating the students table.

- Add a student
  - `INSERT INTO students(name, birthday, gpa, graduate_year) VALUES ('Cantor', '1987-10-22', 3.9, 2009);`
- Delete students who graduated before 1985.
  - `DELETE FROM students WHERE graduate_year < 1985;`
- Update the GPA and graduating year of the student with id 28 to 3.14 and 2015, respectively.
  - `UPDATE students SET gpa = 3.14, graduate_year = 2015 WHERE id = 28;`

The following query returns students with a GPA greater than 3.5, sorted by GPA. It includes each student's name, GPA, and graduating year. `SELECT gpa, name, graduate_year FROM students WHERE gpa > 3.5 ORDER BY gpa DESC;`

Now we turn to the second part of the problem. Here is a sample table for faculty advisers.

id	*name*	*title*
1	Church	Dean
2	Tarski	Professor

We add a new column advisor_id to the students table. This is the foreign key on the previous page. As an example, here's how to return students with GPA whose advisor is Church: `SELECT s.name, s.gpa FROM students s, advisors p WHERE s.advisor_id = p.id AND p.name = 'Church';`.

Variant: What is a SQL join? Suppose you have a table of courses and a table of students. How might a SQL join arise naturally in such a database?

Networking

Most software systems today interact with the network, and it's important for developers even higher up the stack to have basic knowledge of networking. Here are networking questions you may encounter in an interview setting.

23.10 IP, TCP, AND HTTP

Explain what IP, TCP, and HTTP are. Emphasize the differences between them.

Solution: IP, TCP, and HTTP are networking protocols—they help move information between two hosts on a network.

IP, the Internet Protocol, is the lowest-level protocol of the three. It is independent of the physical channel (unlike for example, WiFi or wired Ethernet). Its primary focus is on getting

individual packets from one host to another. Each IP packet has a header that includes the source and destination IP address, the length of the packet, the type of the protocol it's layering (TCP and UDP are common), and an error checking code.

IP moves packets through a network consisting of links and routers. A router has multiple input and output ports. It forwards packets through the network using routing tables that tell the router which output port to forward in incoming packet to. Specifically, a routing table maps IP prefixes to output ports. For example, an entry in the routing table may indicate that all packets with destination IP address matching `171.23.*.*` are to be forwarded out the interface port with id 17. IP routers exchange "link state information" to compute the routing tables; this computation is a kind of shortest path algorithm.

TCP, the Transmission Control Protocol, is an end-to-end protocol built on top of IP. It creates a a continuous reliable bidirectional connection. It does this by maintaining state at the two hosts—this state includes IP packets that have been received, as well as sending acknowledgements. It responds to dropped packets by requesting retransmits—each TCP packet has a sequence number. The receiver signals the transmitter to reduce its transmission rate if the drop rate becomes high—it does this by reducing the received window size. TCP also "demultiplexes" incoming data using the concept of port number—this allows multiple applications on a host with a single IP address to work simultaneously.

HTTP, the Hyper Text Transfer Protocol, is built on top of TCP. An HTTP request or response can be viewed as a text string consisting of a header and a body. The HTTP request and responses header specify the type of the data through the Content-Type field. The response header includes a code, which could indicate success ("200"), a variety of failures ("404"—resource not found), or more specialized scenarios ("302"—use cached version). A few other protocol fields are cookies, request size, compression used.

HTTP was designed to enable the world-wide web, and focused on delivering web pages in a single logical transaction. Its generality means that it has grown to provide much more, e.g., it's commonly used to access remote procedures ("services"). An object—which could be a web page, a file, or a service—is addressed using a URL, Uniform Resource Locator, which is a domain name, a path, and optional arguments. When used to implement services, a common paradigm is for the client to send a request using URL arguments, a JSON object, and files to be uploaded, and the server responds with JSON.

23.11 HTTPS

Describe HTTPS operationally, as well as the ideas underlying it.

Solution: In a nutshell, HTTPS is HTTP with SSL encryption. Because of the commercial importance of the Internet, and the prevalence of wireless networks, it's imperative to protect data from eavesdroppers—this is not done automatically by IP/TCP/HTTP.

Conceptually, one way to encrypt a channel is for the communicating parties to XOR each transmitted byte with a secret byte—the receiver then XORs the received bytes with the secret. This scheme is staggeringly weak—an eavesdropper can look at statistics of the transmitted data to easily figure out which of the 256 bytes is the secret. It leaves open the question of how to get the secrete byte to the receiver. Furthermore, a client who initiates a transaction has no way of knowing if someone has hijacked the network and directed traffic from the client to a malicious server.

SSL resolves these issues. First of all, it uses public key cryptography for key exchange—the idea is that the public key is used by the client to encrypt and send a secret key (which can be viewed

as a generalization of the secret byte; 256 bits is the common size) to the server. This secret key is used to scramble the data to be transmitted in a recoverable way. The secret key is generated via hashing a random number. Public key cryptography is based on the fact that it's possible to create a one-way function, i.e., a function that can be made public and is easy to compute in one direction, but hard to invert with the public information about it. Rivest, Shamir, and Adleman introduced a prototypical such function which is based on several number theoretic facts, such as the relative ease with which primes can be created, that computing $a^b \mod c$ is fast, and that factorization is (likely) hard.

Private key cryptography involves a single key which is a shared secret between the sender and recipient. Private key cryptography offers many orders of magnitude higher bandwidth than public key cryptography. For this reason, once the secret key has been exchanged via public key cryptography, HTTPS turns to private key cryptography, specifically AES. AES operates on 128 bit blocks of data. (If the data to be transmitted is less than 128 bits, it's padded up; if it's greater than 128 bits, it's broken down into 128 bit chunks.) The 128 bit block is represented as 16 bytes in a 4×4 matrix. The secret key is used to select an ordering in which to permute the bytes; it is also used to XOR the bits. This operation is reversible using the secret key.

A small shortcoming of the scheme described above is that it is susceptible to replay attacks—an adversary may not know the exact contents of a transmission, but may know that it represents an action ("add 10 gold coins to my World-Of-Warcraft account"). This can be handled by having the server send a random number which must accompany client requests.

HTTPS adds another layer of security—SSL certificates. The third party issuing the SSL certificate verifies that the HTTPS server is on the domain that it claims to be on, which precludes someone who has hijacked the network from pretending to be the server being requested by the client. This check is orthogonal to the essence of SSL (private key exchange and AES), and it can be disabled, e.g., for testing a server locally.

23.12 DNS

Describe DNS, including both operational as well as implementation aspects.

Solution: DNS is the system which translates domain names—`www.google.com`—to numerical IP addresses—`216.58.218.110`. It is essential to the functioning of the Internet. DNS is what makes it possible for a browser to access a website—the routers in the Internet network rely on IP addresses to lookup next-hops for packets.

There are many challenges to implementing DNS. Here are some of the most important ones.

- There are hundreds of millions of domains.
- There are hundreds of billions of DNS lookups a day.
- Domains and IP addresses are added and updated continuously.

To solve these problems, DNS is implemented as a distributed directory service. A DNS lookup is addressed to a DNS server. Each DNS server stores a database of domain names to IP addresses; if it cannot find a domain name being queried in this database it forwards the request to other DNS servers. These requests can get forwarded all the way up to the root name servers, which are responsible for top-level domains.

At the time of writing there are 13 root name servers. This does not mean there are exactly 13 physical servers. Each operator uses redundancy to provide reliable service. Additionally, these servers are replicated across multiple locations. Key to performance is the heavy use of caching, in the client itself, as well as in the DNS servers.

Part IV

The Honors Class

CHAPTER

24 Honors Class

The supply of problems in mathematics is inexhaustible, and as soon as one problem is solved numerous others come forth in its place.

— "Mathematical Problems," D. HILBERT, 1900

This chapter contains problems that are more difficult to solve than the ones presented earlier. Many of them are commonly asked at interviews, albeit with the expectation that the candidate will not deliver the best solution.

There are several reasons why we included these problems:

- Although mastering these problems is not essential to your success, if you do solve them, you will have put yourself into a very strong position, similar to training for a race with ankle weights.
- If you are asked one of these questions or a variant thereof, and you successfully derive the best solution, you will have strongly impressed your interviewer. The implications of this go beyond being made an offer—your salary, position, and status will all go up.
- Some of the problems are appropriate for candidates interviewing for specialized positions, e.g., optimizing large scale distributed systems, machine learning for computational finance, etc. They are also commonly asked of candidates with advanced degrees.
- Finally, if you love a good challenge, these problems will provide it!

You should be happy if your interviewer asks you a hard question—it implies high expectations, and is an opportunity to shine, compared, for example, to being asked to write a program that tests if a string is palindromic.

Problems roughly follow the sequence of topics of the previous chapters, i.e., we begin with primitive types, and end with graphs. We recommend you solve these problems in a randomized order. The following problems are a good place to begin with: Problems 24.1, 24.6, 24.8, 24.9, 24.12, 24.13, 24.19, 24.20, 24.23, 24.30. White ninja () problems are though challenging, should be solvable by a good candidate given enough time. Black ninja () problems are exceptionally difficult, and are suitable for testing a candidate's response to stress, as described on Page17.

INTRACTABILITY

Coding problems asked in interviews almost always have one (or more) solutions that are fast, e.g., $O(n)$ time complexity. Sometimes, you may be asked a problem that is computationally intractable—there may not exist an efficient algorithm for it. (Such problems are common in the real world.) There are a number of approaches to solving intractable problems:

- brute-force solutions, including dynamic programming, which have exponential time complexity, may be acceptable, if the instances encountered are small, or if the specific parameter that the complexity is exponential in is small;
- search algorithms, such as backtracking, branch-and-bound, and hill-climbing, which prune much of the complexity of a brute-force search;
- approximation algorithms which return a solution that is provably close to optimum;
- heuristics based on insight, common case analysis, and careful tuning that may solve the problem reasonably well;
- parallel algorithms, wherein a large number of computers can work on subparts simultaneously.

24.1 Compute the greatest common divisor

The greatest common divisor (GCD) of nonnegative integers x and y is the largest integer d such that d divides x evenly, and d divides y evenly, i.e., $x \bmod d = 0$ and $y \bmod d = 0$. (When both x and y are 0, take their GCD to be 0.)

Design an efficient algorithm for computing the GCD of two nonnegative integers without using multiplication, division or the modulus operators.

Hint: Use case analysis: both even; both odd; one even and one odd.

Solution: The straightforward algorithm is based on recursion. If $x = y$, $\text{GCD}(x, y) = x$; otherwise, assume without loss of generality, that $x > y$. Then $\text{GCD}(x, y)$ is the $\text{GCD}(x - y, y)$.

The recursive algorithm based on the above does not use multiplication, division or modulus, but for some inputs it is very slow. As an example, if the input is $x = 2^n$, $y = 2$, the algorithm makes 2^{n-1} recursive calls. The time complexity can be improved by observing that the repeated subtraction amounts to division, i.e., when $x > y$, $\text{GCD}(x, y) = \text{GCD}(y, x \bmod y)$, but this approach uses integer division which was explicitly disallowed in the problem statement.

Here is a fast solution, which is also based on recursion, but does not use general multiplication or division. Instead it uses case-analysis, specifically, the cases of one, two, or none of the numbers being even.

An example is illustrative. Suppose we were to compute the GCD of 24 and 300. Instead of repeatedly subtracting 24 from 300, we can observe that since both are even, the result is $2 \times \text{GCD}(12, 150)$. Dividing by 2 is a right shift by 1, so we do not need a general division operation. Since 12 and 150 are both even, $\text{GCD}(12, 150) = 2 \times \text{GCD}(6, 75)$. Since 75 is odd, the GCD of 6 and 75 is the same as the GCD of 3 and 75, since 2 cannot divide 75. The GCD of 3 and 75 is the GCD of 3 and $75 - 3 = 72$. Repeatedly applying the same logic, $\text{GCD}(3, 72) = \text{GCD}(3, 36) = \text{GCD}(3, 18) = \text{GCD}(3, 9) = \text{GCD}(3, 6) = \text{GCD}(3, 3) = 3$. This implies $\text{GCD}(24, 300) = 2 \times 2 \times 3 = 12$.

More generally, the base case is when the two arguments are equal, in which case the GCD is that value, e.g., $\text{GCD}(12, 12) = 12$, or one of the arguments is zero, in which case the other is the GCD, e.g., $\text{GCD}(0, 6) = 6$.

Otherwise, we check if none, one, or both numbers are even. If both are even, we compute the GCD of the halves of the original numbers, and return that result times 2; if exactly one is even, we half it, and return the GCD of the resulting pair; if both are odd, we subtract the smaller from the

larger and return the GCD of the resulting pair. Multiplication by 2 is trivially implemented with a single left shift. Division by 2 is done with a single right shift.

```
def gcd(x, y):
    if x > y:
        return gcd(y, x)
    elif x == 0:
        return y
    elif not x & 1 and not y & 1:  # x and y are even.
        return gcd(x >> 1, y >> 1) << 1
    elif not x & 1 and y & 1:  # x is even, y is odd.
        return gcd(x >> 1, y)
    elif x & 1 and not y & 1:  # x is odd, y is even.
        return gcd(x, y >> 1)
    return gcd(x, y - x)  # Both x and y are odd.
```

Note that the last step leads to a recursive call with one even and one odd number. Consequently, in every two calls, we reduce the combined bit length of the two numbers by at least one, meaning that the time complexity is proportional to the sum of the number of bits in x and y, i.e., $O(\log x + \log y)$.

24.2 Find the first missing positive entry

Let A be an array of length n. Design an algorithm to find the smallest positive integer which is not present in A. You do not need to preserve the contents of A. For example, if $A = \langle 3, 5, 4, -1, 5, 1, -1 \rangle$, the smallest positive integer not present in A is 2.

Hint: First, find an upper bound for x.

Solution: A brute-force approach is to sort A and iterate through it looking for the first gap in the entries after we see an entry equal to 0. The time complexity is that of sorting, i.e., $O(n \log n)$.

Since all we want is the smallest positive number in A, we explore other algorithms that do not rely on sorting. We could store the entries in A in a hash table S (Chapter 12), and then iterate through the positive integers $1, 2, 3, \ldots$ looking for the first one that is not in S. The time complexity is $O(n)$ to create S, and then $O(n)$ to perform the lookups, since we must find a missing entry by the time we get to $n + 1$ as there are only n entries. Therefore the time complexity is $O(n)$. The space complexity is $O(n)$, e.g., if the entries from A are all distinct positive integers.

The problem statement gives us a hint which we can use to reduce the space complexity. Instead of using an external hash table to store the set of positive integers, we can use A itself. Specifically, if A contains k between 1 and n, we set $A[k-1]$ to k. (We use $k-1$ because we need to use all n entries, including the entry at index 0, which will be used to record the presence of 1.) Note that we need to save the presence of the existing entry in $A[k-1]$ if it is between 1 and n. Because A contains n entries, the smallest positive number that is missing in A cannot be greater than $n + 1$.

For example, let $A = \langle 3, 4, 0, 2 \rangle$, $n = 4$. we begin by recording the presence of 3 by writing it in $A[3-1]$; we save the current entry at index 2 by writing it to $A[0]$. Now $A = \langle 0, 4, 3, 2 \rangle$. Since 0 is outside the range of interest, we advance to $A[1]$, i.e., 4, which is within the range of interest. We write 4 in $A[4-1]$, and save the value at that location to index 1, and A becomes $\langle 0, 2, 3, 4 \rangle$. The value at $A[1]$ already indicates that a 2 is present, so we advance. The same holds for $A[2]$ and $A[3]$.

Now we make a pass through A looking for the first index i such that $A[i] \neq i + 1$; this is the smallest missing positive entry, which is 1 for our example.

```
def find_first_missing_positive(A):
    # Record which values are present by writing A[i] to index A[i] - 1 if
    # A[i] is between 1 and len(A), inclusive. We save the value at index A[i]
    # - 1 by swapping it with the entry at i. If A[i] is negative or greater
    # than n, we just advance i.
    for i in range(len(A)):
        while 1 <= A[i] <= len(A) and A[i] != A[A[i] - 1]:
            A[A[i] - 1], A[i] = A[i], A[A[i] - 1]

    # Second pass through A to search for the first index i such that A[i] !=
    # i+1, indicating that i + 1 is absent. If all numbers between 1 and
    # len(A) are present, the smallest missing positive is len(A) + 1.
    return next((i + 1 for i, a in enumerate(A) if a != i + 1), len(A) + 1)
```

The time complexity is $O(n)$, since we perform a constant amount of work per entry. Because we reuse A, the space complexity is $O(1)$.

24.3 Buy and sell a stock k times

This problem generalizes the buy and sell problem introduced on Page 1.

Write a program to compute the maximum profit that can be made by buying and selling a share k times over a given day range. Your program takes k and an array of daily stock prices as input.

Solution: Here is a straightforward algorithm. Iterate over j from 1 to k and iterate through A, recording for each index i the best solution for $A[0, i]$ with j pairs. We store these solutions in an auxiliary array of length n. The overall time complexity will be $O(kn^2)$; by reusing the arrays, we can reduce the additional space complexity to $O(n)$.

We can improve the time complexity to $O(kn)$, and the additional space complexity to $O(k)$ as follows. Define B_i^j to be the most money you can have if you must make $j-1$ buy-sell transactions prior to i and buy at i. Define S_i^j to be the maximum profit achievable with j buys and sells with the jth sell taking place at i. Then the following mutual recurrence holds:

$$\begin{aligned} S_i^j &= A[i] + \max_{i' < i} B_{i'}^j \\ B_i^j &= \max_{i' < i} S_{i'}^{j-1} - A[i] \end{aligned}$$

The key to achieving an $O(kn)$ time bound is the observation that computing B and S requires computing $\max_{i' < i} B_{i'}^{j-1}$ and $\max_{i' < i} S_{i'}^{j-1}$. These two quantities can be computed in constant time for each i and j with a conditional update. In code:

```
def buy_and_sell_stock_k_times(prices, k):
    if not k:
        return 0.0
    elif 2 * k >= len(prices):
        return sum(max(0, b - a) for a, b in zip(prices[:-1], prices[1:]))
    min_prices, max_profits = [float('inf')] * k, [0] * k
    for price in prices:
        for i in reversed(list(range(k))):
```

```
        max_profits[i] = max(max_profits[i], price - min_prices[i])
        min_prices[i] = min(min_prices[i],
                            price - (0 if i == 0 else max_profits[i - 1]))
    return max_profits[-1]
```

Variant: Write a program that determines the maximum profit that can be obtained when you can buy and sell a single share an unlimited number of times, subject to the constraint that a buy must occur more than one day after the previous sell.

24.4 Compute the maximum product of all entries but one

Suppose you are given an array A of integers, and are asked to find the largest product that can be made by multiplying all but one of the entries in A. (You cannot use an entry more than once.) For example, if $A = \langle 3, 2, 5, 4 \rangle$, the result is $3 \times 5 \times 4 = 60$, if $A = \langle 3, 2, -1, 4 \rangle$, the result is $3 \times 2 \times 4 = 24$, and if $A = \langle 3, 2, -1, 4, -1, 6 \rangle$, the result is $3 \times -1 \times 4 \times -1 \times 6 = 72$.

One approach is to form the product P of all the elements, and then find the maximum of $P/A[i]$ over all i. This takes $n - 1$ multiplications (to form P) and n divisions (to compute each $P/A[i]$). Suppose because of finite precision considerations we cannot use a division-based approach; we can only use multiplications. The brute-force solution entails computing all n products of $n - 1$ elements; each such product takes $n - 2$ multiplications, i.e., $O(n^2)$ time complexity.

Given an array A of length n whose entries are integers, compute the largest product that can be made using $n - 1$ entries in A. You cannot use an entry more than once. Array entries may be positive, negative, or 0. Your algorithm cannot use the division operator, explicitly or implicitly.

Hint: Consider the products of the first $i - 1$ and the last $n - i$ elements. Alternatively, count the number of negative entries and zero entries.

Solution: The brute-force approach to compute $P/A[i]$ is to multiplying the entries appearing before i with those that appear after i. This leads to $n(n - 2)$ multiplications, since for each term $P/A[i]$ we need $n - 2$ multiplications.

Note that there is substantial overlap in computation when computing $P/A[i]$ and $P/A[i + 1]$. In particular, the product of the entries appearing before $i + 1$ is $A[i]$ times the product of the entries appearing before i. We can compute all products of entries before i with $n - 1$ multiplications, and all products of entries after i with $n - 1$ multiplications, and store these values in an array of prefix products, and an array of suffix products. The desired result then is the maximum prefix-suffix product. A slightly more space efficient approach is to build the suffix product array and then iterate forward through the array, using a single variable to compute the prefix products. This prefix product is multiplied with the corresponding suffix product and the maximum prefix-suffix product is updated if required.

```
def find_biggest_n_minus_one_product(A):
    # Builds suffix products.
    suffix_products = list(
        reversed(list(itertools.accumulate(reversed(A), operator.mul))))

    # Finds the biggest product of (n - 1) numbers.
    prefix_product, max_product = 1, float('-inf')
```

```
    for i in range(len(A)):
        suffix_product = suffix_products[i + 1] if i + 1 < len(A) else 1
        max_product = max(max_product, prefix_product * suffix_product)
        prefix_product *= A[i]
    return max_product
```

The time complexity is $O(n)$; the space complexity is $O(n)$, since the solution uses a single array of length n.

We now solve this problem in $O(n)$ time and $O(1)$ additional storage. The insight comes from the fact that if there are no negative entries, the maximum product comes from using all but the smallest element. (Note that this result is correct if the number of 0 entries is zero, one, or more.)

If the number of negative entries is odd, regardless of how many 0 entries and positive entries, the maximum product uses all entries except for the negative entry with the smallest absolute value, i.e., the least negative entry.

Going further, if the number of negative entries is even, the maximum product again uses all but the smallest nonnegative element, assuming the number of nonnegative entries is greater than zero. (This is correct, even in the presence of 0 entries.)

If the number of negative entries is even, and there are no nonnegative entries, the result must be negative. Since we want the largest product, we leave out the entry whose magnitude is largest, i.e., the greatest negative entry.

This analysis yields a two-stage algorithm. First, determine the applicable scenario, e.g., are there an even number of negative entries? Consequently, perform the actual multiplication to get the result.

```
def find_biggest_n_minus_one_product(A):
    number_of_negatives = 0
    least_nonnegative_idx = least_negative_idx = greatest_negative_idx = None

    # Identify the least negative, greatest negative, and least nonnegative
    # entries.
    for i, a in enumerate(A):
        if a < 0:
            number_of_negatives += 1
            if least_negative_idx is None or A[least_negative_idx] < a:
                least_negative_idx = i
            if greatest_negative_idx is None or a < A[greatest_negative_idx]:
                greatest_negative_idx = i
        else:  # a >= 0.
            if least_nonnegative_idx is None or a < A[least_nonnegative_idx]:
                least_nonnegative_idx = i

    idx_to_skip = (least_negative_idx
                   if number_of_negatives % 2 else least_nonnegative_idx if
                   least_nonnegative_idx is not None else greatest_negative_idx)
    return functools.reduce(
        lambda product, a: product * a,
        # Use a generator rather than list comprehension to
        # avoid extra space.
        (a for i, a in enumerate(A) if i != idx_to_skip),
        1)
```

The algorithm performs a traversal of the array, with a constant amount of computation per entry, a nested conditional, followed by another traversal of the array, with a constant amount of computation per entry. Hence, the time complexity is $O(n) + O(1) + O(n) = O(n)$. The additional space complexity is $O(1)$, corresponding to the local variables.

Variant: Let A be as above. Compute an array B where $B[i]$ is the product of all elements in A except $A[i]$. You cannot use division. Your time complexity should be $O(n)$, and you can only use $O(1)$ additional space.

Variant: Let A be as above. Compute the maximum over the product of all triples of distinct elements of A.

24.5 Compute the longest contiguous increasing subarray

An array is increasing if each element is less than its succeeding element except for the last element.

Implement an algorithm that takes as input an array A of n elements, and returns the beginning and ending indices of a longest increasing subarray of A. For example, if $A = \langle 2, 11, 3, 5, 13, 7, 19, 17, 23 \rangle$, the longest increasing subarray is $\langle 3, 5, 13 \rangle$, and you should return $(2, 4)$.

Hint: If $A[i] \geq A[i+1]$, instead of checking $A[i+1] \leq A[i+2]$, go further out in the array.

Solution: The brute-force algorithm is to test every subarray: two nested loops iterate through the starting and ending index of the subarray, an inner loop to test if the subarray is increasing, and some logic to track the longest subarray. The time complexity is $O(n^3)$, e.g., for the array $\langle 0, 1, 2, 3, \ldots, n-1 \rangle$. The time complexity can easily be improved to $O(n^2)$ by caching whether the subarray $A[i, j]$ is increasing when examining $A[i, j+1]$.

Looking more carefully at the example, the longest subarray ending at 3 is $A[2, 2]$, because $3 < 11$. Conversely, since $13 > 5$, the longest subarray ending at 13 is the longest subarray ending at 5 (which is $A[2, 3]$) together with 13, i.e., $A[2, 4]$. In general, the longest increasing subarray ending at index $j+1$ inclusive is

(1.) the single entry $A[j+1]$, if $A[j+1] \leq A[j]$, or

(2.) the longest increasing subarray ending at index j together with $A[j+1]$, if $A[j+1] > A[j]$.

This fact can be used directly to improve the brute-force algorithm to one whose time complexity is $O(n)$.

The additional space complexity is $O(1)$, since all we need is the length of the longest subarray ending at j when processing $j+1$. Two additional variables can be used to hold the length and ending index of the longest increasing subarray seen so far.

We can heuristically improve upon the $O(n)$ algorithm by observing that if $A[i-1] \not< A[i]$ (i.e., we are starting to look for a new subarray starting at i) and the longest contiguous subarray seen up to index i has length L, we can move on to index $i + L$ and work backwards towards i. Specifically, if for any $j, i < j \leq i + L$ we have $A[j-1] \not< A[j]$, we can skip the earlier indices. For example, after processing 13, we work our way back from the entry at index $4 + 3 = 7$, i.e., 13's index plus the length of the longest increasing subarray seen so far (3). Since $A[7] = 17 < A[6] = 19$, we do not need to examine prior entries—no increasing array ending at $A[7]$ can be longer than the current best.

```
Subarray = collections.namedtuple('Subarray', ('start', 'end'))


def find_longest_increasing_subarray(A):
    result = Subarray(0, 0)
    i, max_length = 0, 1
    while i < len(A) - max_length:
        # Backward check and skip if A[j] >= A[j + 1].
        for j in range(i + max_length, i, -1):
            if A[j - 1] >= A[j]:
                i = j
                break
        else:  # Forward check if it is not skippable (the loop ended normally)
            i += max_length
            while i < len(A) and A[i - 1] < A[i]:
                i, max_length = i + 1, max_length + 1
            result = Subarray(i - max_length, i - 1)
    return result
```

Skipping is a heuristic in that it does not improve the worst-case complexity. If the array consists of alternating 0s and 1s, we still examine each element, implying an $O(n)$ time bound, but the best-case complexity reduces to $O(\max(n/L, L))$, where L is the length of the longest increasing subarray.

24.6 Rotate an array

Let A be an array of n elements. If memory is not a concern, rotating A by i positions is trivial; we create a new array B of length n, and set $B[j] = A[(i+j) \bmod n]$ for each j. If all we have is additional storage for c elements, we can repeatedly rotate the array by c a total of $\lceil i/c \rceil$ times; this increases the time complexity to $O(n\lceil i/c \rceil)$.

Design an algorithm for rotating an array A of n elements to the right by i positions. Do not use library functions implementing rotate.

Hint: Use concrete examples to form a hypothesis relating n, i, and the number of cycles.

Solution: There are two brute-force algorithms: perform shift-by-one i times, which has $O(ni)$ time complexity and $O(1)$ space complexity. The other is to use an additional array of length i as a buffer to move elements i at a time. This has $O(n)$ time complexity and $O(i)$ space complexity.

The key to achieving both $O(n)$ time complexity and $O(1)$ space complexity is to use the fact that a permutation can be applied using constant additional storage (Problem 5.10 on Page 50) with the permutation corresponding to a rotation.

A rotation by itself is not a cyclic permutation. However, a rotation is a permutation, and as such can be decomposed to a set of cyclic permutations. For example, for the case where $n = 6$ and $i = 2$, the rotation corresponds to the permutation $\langle 4, 5, 1, 2, 3, 4 \rangle$. This permutation can be achieved by the cyclic permutations $(0, 2, 4)$ and $(1, 3, 5)$. Similarly, when $n = 15$ and $i = 6$, the cycles are $\langle 0, 6, 12, 3, 9 \rangle$, $\langle 1, 7, 13, 4, 10 \rangle$, and $\langle 2, 8, 14, 5, 11 \rangle$.

The examples lead us to conjecture the following:

(1.) All cycles have the same length, and are a shifted version of the cycle $\langle 0, i \bmod n, 2i \bmod n, \ldots, (l-1)i \bmod n \rangle$.

(2.) The number of cycles is the GCD of n and i.

These conjectures can be justified on heuristic grounds. (A formal proof requires looking at the prime factorizations for i and n.)

Assuming these conjectures to be correct, we can apply the rotation one cycle at a time, as follows. The first elements of the different cyclic permutations are at indices $0, 1, 2, \ldots, \text{GCD}(n, i) - 1$. For each cycle, we apply it by shifting elements in the cycle one-at-a-time.

```
def rotate_array(rotate_amount, A):
    def apply_cyclic_permutation(rotate_amount, offset):
        temp = A[offset]
        for i in range(1, cycle_length):
            idx = (offset + i * rotate_amount) % len(A)
            A[idx], temp = temp, A[idx]
        A[offset] = temp

    rotate_amount %= len(A)
    if rotate_amount == 0:
        return
    num_cycles = fractions.gcd(len(A), rotate_amount)
    cycle_length = len(A) // num_cycles
    for c in range(num_cycles):
        apply_cyclic_permutation(rotate_amount, c)
```

The time complexity is $O(n)$, since we perform a constant amount of work per entry. The space complexity is $O(1)$.

We now provide an alternative to the permutation approach. The new solution works well in practice and is considerably simpler. Assume that $A = \langle 1, 2, 3, 4, a, b \rangle$, and $i = 2$. Then in the rotated A there are two subarrays, $\langle 1, 2, 3, 4 \rangle$ and $\langle a, b \rangle$ that keep their original orders. Therefore, rotation can be seen as the exchanges of the two subarrays of A. It is easy to perform these exchanges in $O(n)$ time. To implement these exchanges to use $O(1)$ space we use an array-reverse function. Using A and i as an example, we first reverse A to get A' ($\langle 1, 2, 3, 4, a, b \rangle$ becomes $\langle b, a, 4, 3, 2, 1 \rangle$), then reverse the first i elements of A' ($\langle b, a, 4, 3, 2, 1 \rangle$ becomes $\langle a, b, 4, 3, 2, 1 \rangle$), and reverse the remaining elements starting from the ith element of A' ($\langle a, b, 4, 3, 2, 1 \rangle$ becomes $\langle a, b, 1, 2, 3, 4 \rangle$) which yields the rotated A.

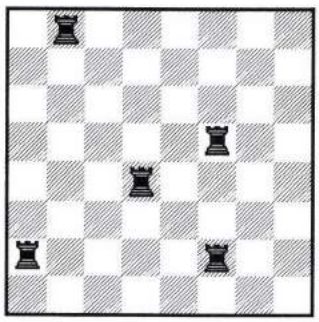
```
def rotate_array(i, A):
    i %= len(A)

    def reverse(begin, end):
        while begin < end:
            A[begin], A[end] = A[end], A[begin]
            begin, end = begin + 1, end - 1

    reverse(0, len(A) - 1)
    reverse(0, i - 1)
    reverse(i, len(A) - 1)


# Although the following function is very natural way to rotate an array,
# its use of sublists leads to copy from original list, and therefore
# linear space complexity.
def rotate_array_naive(i, A):
```

```
i %= len(A)
A[:] = A[::-1]  # reverse whole list
A[:i] = A[:i][::-1]  # reverse A[:i] part
A[i:] = A[i:][::-1]  # reverse A[i:] part
```

We note in passing that a completely different approach is to perform the rotation in blocks of k, reusing freed space for temporary storage. It can be made to work, but the final rotation has to be performed very carefully, and the resulting code is complex.

24.7 Identify positions attacked by rooks

This problem is concerned with computing squares in a chessboard which can be attacked by rooks that have been placed at arbitrary locations. The scenario is illustrated in Figure 24.1(a).



(a) Initial placement of 5 rooks on an 8×8 chessboard.

1	**0**	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	**0**	1	1
1	1	1	**0**	1	1	1	1
1	1	1	1	1	1	1	1
0	1	1	1	1	**0**	1	1
1	1	1	1	1	1	1	1

(b) Rook placement from (a) encoded using an 8×8 2D array—a 0 indicates a rook is placed at that position.

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	1	**0**	1	**0**	1	1
0	**0**	1	**0**	1	**0**	1	1
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	1	**0**	1	**0**	1	1
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	1	**0**	1	**0**	1	1

(c) 2D array encoding positions that can be attacked by rooks placed as in (a)—a 0 indicates an attacked position.

Figure 24.1: Rook attack.

Write a program which takes as input a 2D array A of 1s and 0s, where the 0s encode the positions of rooks on an $n \times m$ chessboard, as show in Figure 24.1(b) and updates the array to contain 0s at all locations which can be attacked by rooks, as shown in Figure 24.1(c).

Hint: Make use of the first row and the first column.

Solution: This problem is trivial with an additional n-bit array R and an additional m-bit array C. We simply initialize R and C to 1s. Then we iterate through all entries of A, and for each (i, j) such that $A[i][j] = 0$, we set $R[i]$ and $C[j]$ to 0. Consequently, a 0 in $R[i]$ indicates that Row i should be set to 0; columns are analogous. A second iteration through all entries can be used to set the 0s in A.

The drawback with the above approach is the use of $O(n + m)$ additional storage. The solution is to use storage in A itself. The reason we can do this is because if a single 0 appears in a row, the entire row is cleared. Consequently, we can store a single extra bit r denoting whether Row 0 has a 0 within it. Now Row 0 can play the role of C in the algorithm in the previous paragraph. If we record a 0 in $R[i]$, that is the value we would be writing at that location, so the original entry is not lost. If $R[i]$ holds a 1 after the first pass, it retains that value, unless r indicates Row 0 is to be cleared. Columns are handled in exactly the same way.

```
def rook_attack(A):
    m, n = len(A), len(A[0])
    has_first_row_zero = any(not A[0][j] for j in range(n))
    has_first_column_zero = any(not A[i][0] for i in range(m))

    for i in range(1, m):
        for j in range(1, n):
            if not A[i][j]:
                A[i][0] = A[0][j] = 0

    for i in range(1, m):
        if not A[i][0]:
            for j in range(1, n):
                A[i][j] = 0

    for j in range(1, n):
        if not A[0][j]:
            for i in range(1, m):
                A[i][j] = 0

    if has_first_row_zero:
        for j in range(n):
            A[0][j] = 0

    if has_first_column_zero:
        for i in range(m):
            A[i][0] = 0
```

The time complexity is $O(nm)$ since we perform $O(1)$ computation per entry. The space complexity is $O(1)$.

24.8 JUSTIFY TEXT

This problem is concerned with justifying text. It abstracts a problem arising in typesetting. The input is specified as a sequence of words, and the target line length. After justification, each individual line must begin with a word, and each subsequent word must be separated from prior words with at least one blank. If a line contains more than one word, it should not end in a blank. The sequences of blanks within each line should be as close to equal in length as possible, with the longer blank sequences, if any, appearing at the initial part of the line. As an exception, the very last line should use single blanks as separators, with additional blanks appearing at its end.

For example, if $A = \langle$"The", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dogs."$\rangle$ and the line length L is 11, then the returned result should be "The␣␣␣quick", "brown␣␣␣fox", "jumped␣over", "the␣␣␣␣␣lazy", "dogs.␣␣␣␣␣␣". The symbol ␣ denotes a blank.

Write a program which takes as input an array A of strings and a positive integer L, and computes the justification of the text specified by A.

Hint: Solve it on a line-by-line basis, assuming a single blank between pairs of words. Then figure out how to distribute excess blanks.

Solution: The challenge in solving this problem is that it requires lookahead. Specifically, the number of blanks between words in a line cannot be computed till complete set of words in that line is known.

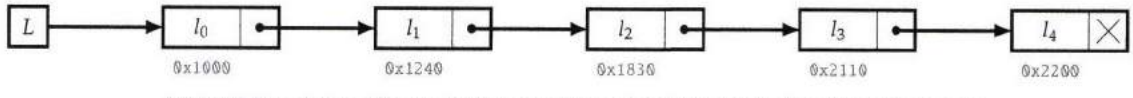
We solve the problem on a line-by-line basis. First, we compute the words that go into each line, assuming a single blank between words. After we know the words in a line, we compute the number of blanks in that line and distribute the blanks evenly. The final line is special-cased.

```
def justify_text(words, L):
    curr_line_length, result, curr_line = 0, [], []
    for word in words:
        if curr_line_length + len(word) + len(curr_line) > L:
            # Distribute equally between words in curr_line.
            for i in range(L - curr_line_length):
                curr_line[i % (len(curr_line) - 1 or 1)] += ' '
            result.append(''.join(curr_line))
            curr_line, curr_line_length = [], 0
        curr_line.append(word)
        curr_line_length += len(word)
    # Use ljust(L) to pad the last line with the appropriate number of blanks.
    return result + [' '.join(curr_line).ljust(L)]
```

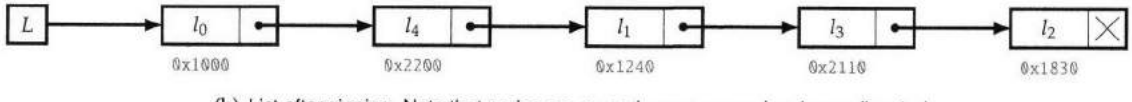
Let n be the sum of the lengths of the strings in A. We spend $O(1)$ time per character in the first pass as well as the second pass, yielding an $O(n)$ time complexity.

24.9 Implement list zipping

Let L be a singly linked list. Assume its nodes are numbered starting at 0. Define the zip of L to be the list consisting of the interleaving of the nodes numbered $0, 1, 2, \ldots$ with the nodes numbered $n-1, n-2, n-3, \ldots$, where n is the number of nodes in the list. The zipping function is illustrated in Figure 24.2.



(a) List before zipping. The number in hex below each node represents its address in memory.



(b) List after zipping. Note that nodes are reused—no memory has been allocated.

Figure 24.2: Zipping a list.

Implement the zip function.

Hint: Consider traversing the list in reverse order.

Solution: A brute-force approach is to iteratively identify the head and tail, remove them from the original list, and append the pair to the result. The time complexity is $O(n)+O(n-2)+O(n-4)+\cdots = O(n^2)$, where n is the number of nodes. The space complexity is $O(1)$.

The $O(n^2)$ complexity comes from having to repeatedly traverse the list to identify the tail. Note that getting the head of a singly linked list is an $O(1)$ time operation. This suggests paying a one-time cost of $O(n)$ to reverse the second half of the original list. Now all we need to do is interleave this with the first half of the original list.

```
def zipping_linked_list(L):
    if not L or not L.next:
        return L

    # Finds the second half of L.
    slow = fast = L
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next

    first_half_head = L
    second_half_head = slow.next
    slow.next = None  # Splits the list into two lists.

    second_half_head = reverse_linked_list(second_half_head)

    # Interleave the first half and the reversed of the second half.
    first_half_iter, second_half_iter = first_half_head, second_half_head
    while second_half_iter:
        second_half_iter.next, first_half_iter.next, second_half_iter = (
            first_half_iter.next, second_half_iter, second_half_iter.next)
        first_half_iter = first_half_iter.next.next
    return first_half_head
```

The time complexity is $O(n)$. The space complexity is $O(1)$.

24.10 Copy a postings list

A postings list is a singly linked list with an additional "jump" field at each node. The jump field points to any other node. Figure 24.3 illustrates a postings list with four nodes.

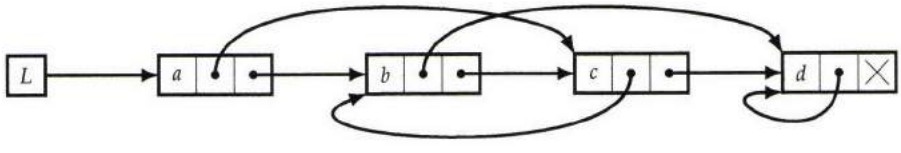


Figure 24.3: A postings list.

Implement a function which takes a postings list and returns a copy of it. You can modify the original list, but must restore it to its initial state before returning.

Hint: Copy the jump field and then copy the next field.

Solution: Here is a brute-force algorithm. First, create a copy of the postings list, without assigning values to the jump field. Next, use a hash table to store the mapping from nodes in the original postings list to nodes in the copied list. Finally, traverse the original list and the new list in tandem,

using the mapping to assign each jump field. The time and space complexity are $O(n)$, where n is the number of nodes in the original postings list.

The key to improving space complexity is to use the next field for each node in the original list to record the mapping from the original node to its copy. To avoid losing the structure of the original list, we use the next field in each copied node to point to the successor of its original node. See Figure 24.4(b) for an example. Now we can proceed like we did in the brute-force algorithm. We assign the jump field in the copied nodes, using the next field in the original list to get the corresponding nodes in the copy. See Figure 24.4(c). Finally, we undo the changes made to the original list and update the next fields of the copied list, as in Figure 24.4(d) for an example.

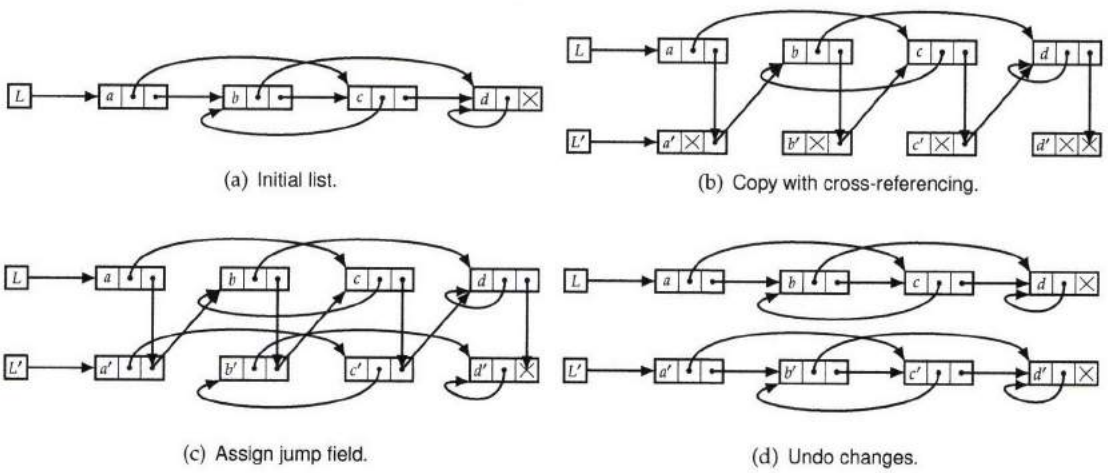


Figure 24.4: Duplicating a postings list.

```
def copy_postings_list(L):
    if not L:
        return None

    # Stage 1: Makes a copy of the original list without assigning the jump
    #          field, and creates the mapping for each node in the original
    #          list to the copied list.
    it = L
    while it:
        new_node = PostingListNode(it.order, it.next, None)
        it.next = new_node
        it = new_node.next

    # Stage 2: Assigns the jump field in the copied list.
    it = L
    while it:
        if it.jump:
            it.next.jump = it.jump.next
        it = it.next.next

    # Stage 3: Reverts the original list, and assigns the next field of
    #          the copied list.
    it = L
```

```
    new_list_head = it.next
    while it.next:
        it.next, it = it.next.next, it.next
    return new_list_head
```

The time complexity is $O(n)$. The space complexity is $O(1)$.

24.11 Compute the longest substring with matching parens

Problem 8.3 on Page 102 defines matched strings of parens, brackets, and braces. This problem is restricted to strings of parens. Specifically, this problem is concerned with a long substrings of matched parens. As an example, if s is "(((())()(()(", then "(())()" is a longest substring of matched parens.

Write a program that takes as input a string made up of the characters '(' and ')', and returns the size of a maximum length substring in which the parens are matched.

Hint: Start with a brute-force algorithm and then refine it by considering cases in which you can advance more quickly.

Solution: One approach would be to run the algorithm in Solution 8.3 on Page 102 on all substrings. The time complexity is $O(n^3)$, where n is the length of the string—there are $\binom{n}{2} = \frac{n(n-1)}{2}$ substrings, and the matching algorithm runs in time $O(n)$.

Note that if a prefix of a string fails the matched test because of an unmatched right parens, no extension of that prefix can be matched. Therefore, a faster approach is for each i to find the longest substring starting at the ith character that is matched. This leads to an $O(n^2)$ algorithm.

Finally, if a substring ends in an unmatched right parens, but all of that substring's prefixes ending in right parens are matched, then no nonempty suffix of the prefix can be the prefix of a matched string, since any such suffix has fewer left parens. Therefore, as soon as a prefix has an unmatched right parens, we can continue with the next character after that prefix. We store the left parentheses' indices in a stack. At the same time, when we process a right parens for the given prefix, if it matched, we use the index at the top of the stack to update the longest matched string seen for this prefix.

For the given example, "(((())()(()(", we push left parentheses and pop on right parentheses. Before the first pop, the stack is $\langle 0, 1, 2 \rangle$, where the first array element is the bottom of the stack. The corresponding matched substring length is $3 - 1 = 2$. Before the second pop, the stack is $\langle 0, 1 \rangle$. The corresponding matched substring length is $4 - 0 = 4$. Before the third pop, the stack is $\langle 0, 5 \rangle$. The corresponding matched substring length is $6 - 0 = 6$. Before the last pop, the stack is $\langle 0, 7, 8 \rangle$. The corresponding matched substring length is $9 - 7 = 2$.

```
def longest_matching_parentheses(s):
    max_length, end, left_parentheses_indices = 0, -1, []
    for i, c in enumerate(s):
        if c == '(':
            left_parentheses_indices.append(i)
        elif not left_parentheses_indices:
            end = i
        else:
            left_parentheses_indices.pop()
```

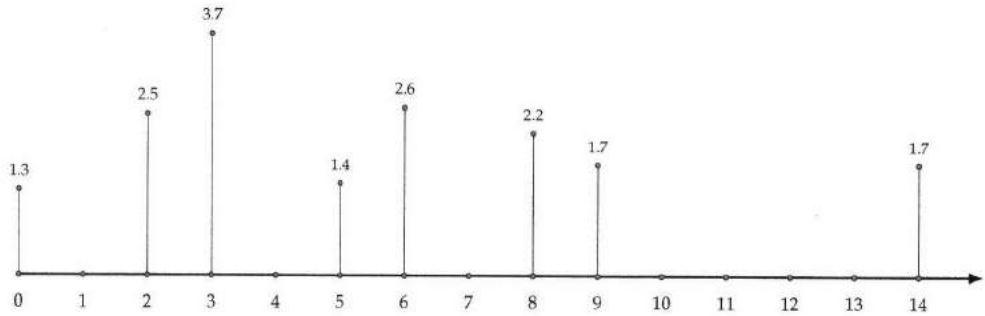
```
            start = (left_parentheses_indices[-1]
                     if left_parentheses_indices else end)
            max_length = max(max_length, i - start)
    return max_length
```

The time and space complexity are $O(n)$.

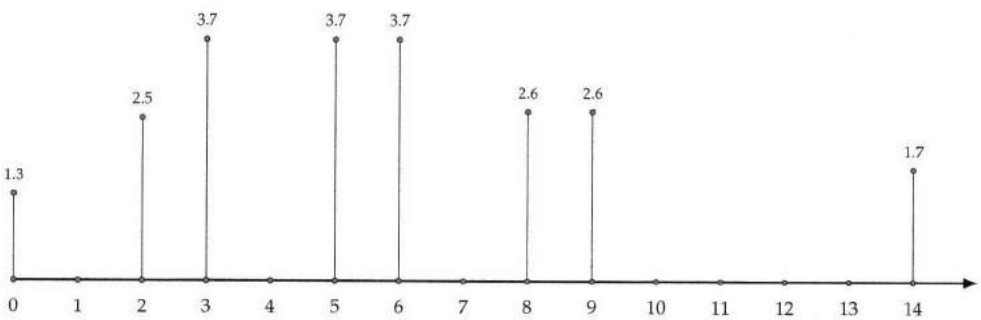
Variant: Solve the same problem using only $O(1)$ space.

24.12 Compute the maximum of a sliding window

Network traffic control sometimes requires studying traffic volume over time. This problem explores the development of an efficient algorithm for computing the traffic volumes.



(a) Traffic at various timestamps.



(b) Maximum traffic over a window size of 3.

Figure 24.5: Traffic profile before and after windowing.

You are given traffic at various timestamps and a window length. Compute for each input timestamp, the maximum traffic over the window length time interval which ends at that timestamp. See Figure 24.5 for an example.

Hint: You need to be able to identify the maximum element in a queue quickly.

Solution: Assume the input is specified by window length w and an array A of pairs consisting of integer timestamp and corresponding traffic volume. If A is not sorted by timestamp, we sort it. For example, the traffic in 24.5(a) corresponds to the array $\langle(0, 1.3), (2, 2.5), (3, 3.7), (5, 1.4), (6, 2.6), (8, 2.2), (9, 1.7), (14, 1.1)\rangle$.

The brute-force algorithm entails finding for each input timestamp, the maximum in the subarray consisting of elements whose timestamps lie in the window ending at that timestamp. The time complexity is $O(nw)$, where n is the length of A. The reason is that every window may have up to $w + 1$ elements.

The intuition for improving the time complexity of the brute-force approach stems from noting that as we advance the window only the boundary changes. Specifically, some older elements fall out of the window, and a new element is added. Therefore a queue is a perfect representation for the window. We need the maximum traffic within each window, which suggests using Solution 8.9 on Page 109.

Initialize Q to an empty queue with maximum. Iteratively enqueue (t_i, t_i) in order of increasing i. For each i, iteratively dequeue Q until the difference of the timestamp at Q's head and t_i is less than or equal to w. The sequence of maximum values in the queue for each i is the desired result. For the input in Figure 24.5(a) on the previous page with window size of 3, the output is $\langle(0, 1.3), (2, 2.5), (3, 3.7), (5, 3.7), (6, 3.7), (8, 2.6), (9, 2.6), (14, 1.7)\rangle$. See Figure 24.5(b) on the preceding page for a graphical representation.

```
class TrafficElement:
    def __init__(self, time, volume):
        self.time = time
        self.volume = volume


    # Following operators are needed for QueueWithMax with maximum.
    def __lt__(self, other):
        return self.volume < other.volume

    def __eq__(self, other):
        return self.time == other.time and self.volume == other.volume


def calculate_traffic_volumes(A, w):
    sliding_window = QueueWithMax()
    maximum_volumes = []
    for traffic_info in A:
        sliding_window.enqueue(traffic_info)
        while traffic_info.time - sliding_window.head().time > w:
            sliding_window.dequeue()
        maximum_volumes.append(
            TrafficElement(traffic_info.time, sliding_window.max().volume))
    return maximum_volumes
```

Each element is enqueued once. Each element is dequeued at most once. Since the queue with maximum data structure has an $O(1)$ amortized time complexity per operation, the overall time complexity is $O(n)$. The additional space complexity is $O(w)$.

24.13 Implement a postorder traversal without recursion

This problem is concerned with traversing nodes in a binary tree in postorder fashion. See Page 113 for details and examples of these traversals. Generally speaking, a traversal computation is easy to implement if recursion is allowed.

Write a program which takes as input a binary tree and performs a postorder traversal of the tree. Do not use recursion. Nodes do not contain parent references.

Hint: Study the function call stack for the recursive versions.

Solution: The brute-force approach to remove recursion from a function is to mimic the function call stack with an explicit stack. One of the challenges with this approach is to determine where to return to.

Now we address the problem of implementing a postorder traversal without recursion. First we discuss a roundabout way of doing this. An inverted preorder traversal is the following: visit root, inverted preorder traverse the right subtree, then inverted preorder traverse the left subtree. For example, the inverted preorder traversal of the tree in Figure 9.1 on Page 112 visits nodes in the following order: $\langle A, I, O, P, J, K, N, L, M, B, F, G, H, C, E, D \rangle$.

Intuitively, since the inverted preorder traversal is visit root, traverse right, traverse left, its reverse is traverse left, traverse right, visit root, i.e., the postorder traversal. Therefore, one way to compute the postorder traversal visit sequence without using recursion is to perform an inverted preorder traversal and instead of outputting nodes, we store them. When the inverted preorder traversal is complete, we iterate through the nodes in last-in, first-out order, which gives the postorder traversal sequence.

The inverted preorder traversal itself can be performed nonrecursively using the solution to the first part of this problem, with the order in which the left and right children are pushed swapped. This algorithm is implemented in the code below.

```
def postorder_traversal(tree):
    def inverted_preorder_traversal(tree):
        path_stack, result = [tree], []
        while path_stack:
            curr = path_stack.pop()
            if not curr:
                continue
            result.append(curr.data)
            path_stack.extend([curr.left, curr.right])
        return result

    return inverted_preorder_traversal(tree)[::-1]
```

The time and space complexity are both $O(n)$, where n is the number of nodes in the tree.

In addition to its being unintuitive, a more technical limitation of the approach given above is that it requires $O(n)$ additional space. If the result is to be returned as an array, this is unavoidable. However, if we are simply required to print the nodes, it is possible to reduce the space complexity to $O(h)$ where h is the height of the tree.

We know that a recursive implementation of a postorder traversal takes $O(n)$ and $O(h)$ space, so we should be able to achieve this complexity by using a stack to mimic the function call stack. One challenge is keeping track of where to return to, since there are two recursive calls. The function call stack keeps a return address, but instruction addresses are not accessible from user code. We can determine where to continue from by inspecting where the last visited node is relative to the current node. This is explained in more detail below.

We maintain a stack of nodes which evolves exactly as the sequence of nodes that the recursive algorithm makes calls from.

To determine when a nonleaf at the top of the stack is ready for visiting, we need to know if we are moving back up the tree, and if so, which side of this nonleaf we are returning from. If we are coming back up from the left, we do not want to push the left child again, but do want to push the right child, since we still need to mimic the second recursive call. If we are coming up from the right child both children have been visited, so we want to pop the stack and visit the nonleaf node.

```
# We use stack and previous node pointer to simulate postorder traversal.
def postorder_traversal(tree):
    if not tree:  # Empty tree.
        return []

    path_stack, prev, postorder_sequence = [tree], None, []
    while path_stack:
        curr = path_stack[-1]
        if not prev or prev.left is curr or prev.right is curr:
            # We came down to curr from prev.
            if curr.left:  # Traverse left.
                path_stack.append(curr.left)
            elif curr.right:  # Traverse right.
                path_stack.append(curr.right)
            else:  # Leaf node, visit current node.
                postorder_sequence.append(curr.data)
                path_stack.pop()
        elif curr.left is prev:
            # Done with left, now traverse right.
            if curr.right:
                path_stack.append(curr.right)
            else:  # No right child, so visit curr.
                postorder_sequence.append(curr.data)
                path_stack.pop()
        else:
            # Finished traversing left and right, so visit curr.
            postorder_sequence.append(curr.data)
            path_stack.pop()
        prev = curr
    return postorder_sequence
```

The time complexity is $O(n)$, since we perform a constant amount of work per node (a push and a pop). The space complexity is $O(h)$, since the stack corresponds to a path starting at the root.

24.14 Compute fair bonuses

You manage a team of developers. You have to give concert tickets as a bonus to the developers. For each developer, you know how many lines of code he wrote the previous week, and you want to reward more productive developers. [1]

The developers sit in a row. Each developer, save for the first and last, has two neighbors. You must give each developer one or more tickets in such a way that if a developer has written more

[1] In practice, lines-of-code is a very poor productivity metric.

lines of code than a neighbor, then he receives more tickets than his neighbor. (If two neighboring developers have written an equal number of lines of code, they do not have to receive the same number of tickets.)

For example, if Andy, Bob, Charlie, and David sit in a row from left to right, and they wrote 300, 400, 200, and 500 lines of code, respectively, the previous week, then Andy and Charlie should receive one ticket each, and Bob and David should receive two tickets, for a total of six tickets. If instead they wrote 300, 400, 400, and 400 lines of code, respectively, then Andy, Charlie, and David should receive a single ticket, and Bob should receive two tickets for a total of five tickets.

Your task is to develop an algorithm that computes the minimum number of tickets you need to buy to satisfy the constraint.

Write a program for computing the minimum number of tickets to distribute to the developers, while ensuring that if a developer has written more lines of code than a neighbor, then he receives more tickets than his neighbor.

Hint: Consider iteratively improving an assignment that may not satisfy the constraint.

Solution: A brute-force approach is to start by giving each developer a ticket. Next we perform the following iteration. We check if all developers are satisfied. If they are all satisfied, we are done. Otherwise, if some developer is not satisfied, i.e., he has written more lines of code than a neighbor, but does not have more tickets than that neighbor, we give him one more ticket than his neighbor. This approach uses the minimum number of tickets initially, and every additional ticket that is given is necessary. The time complexity is $O(kn^2)$, where k is the maximum number of tickets given to any single developer, and n is the number of developers.

The key insight to a better algorithm is the observation that the least productive developer never needs to be given more than a single ticket. We can propagate this observation by processing developers in increasing order of productivity. Subsequently, when we process a developer if his neighbors have been processed, he must be at least as productive as them. If a developer is more productive than a neighbor, he must be given at a minimum one more ticket than that neighbor. If a developer is only as productive as his neighbors, we only need give him the same number of tickets, as per the problem specification.

For the given example, our algorithm starts by giving 1 ticket to David. Andy is next in order of productivity, so we give him 1 ticket, since he has only one neighbor, who is more productive than him. Next we process Bob. He is more productive than Andy, so we give him $1 + 1 = 2$ tickets. Then comes Charlie. He is a neighbor of Charlie and David, and is more productive than both, so we give him $\max(2, 1) + 1 = 3$ tickets, for a total of 7 tickets.

This approach yields the correct result because once a developer is processed, we only process developers who are at least as productive in the future, meaning that once his bonus is assigned, it will never need updating in the future. Furthermore, any bonus that we assign is forced upon us by the problem constraints.

A min-heap is a suitable data structure for processing the developers, and is used in the following program.

```
def calculate_bonus(productivity):
    # Stores (productivity, index)-pair in min_heap where ordered by
    # productivity.
    EmployeeData = collections.namedtuple('EmployeeData', ('productivity',
```

```
                                                    'index'))
    min_heap = [EmployeeData(p, i) for i, p in enumerate(productivity)]
    heapq.heapify(min_heap)

    # Initially assigns one ticket to everyone.
    tickets = [1] * len(productivity)
    # Fills tickets from lowest rating to highest rating.
    while min_heap:
        next_dev = heapq.heappop(min_heap)[1]
        # Handles the left neighbor.
        if next_dev > 0 and productivity[next_dev] > productivity[next_dev - 1]:
            tickets[next_dev] = tickets[next_dev - 1] + 1
        # Handles the right neighbor.
        if (next_dev + 1 < len(tickets)
                and productivity[next_dev] > productivity[next_dev + 1]):
            tickets[next_dev] = max(tickets[next_dev],
                                    tickets[next_dev + 1] + 1)
    return sum(tickets)
```

Since each extraction from a min-heap takes time $O(\log n)$, the time complexity is $O(n \log n)$.

The approach presented above is in the spirit of a brute-force solution. On some reflection, a total ordering on the developers is overkill, since the specified constraint is very local. Indeed, we can improve the time complexity to $O(n)$ by making two passes over the array.

We start by giving each developer a single ticket. Then we make a left-to-right pass in which we give each developer who has more productivity than the developer on his left one ticket more than the developer on his left. We then do the same in a right-to-left pass.

Any amount added is required, so we cannot get by with fewer tickets. Note that every developer who is more productive than his right neighbor has more tickets than that neighbor. Furthermore, if a developer is more productive than his left neighbor, in the left-to-right pass we already give him more tickets than his left neighbor, and we can only increase his ticket count in the right-to-left pass.

```
def calculate_bonus(productivity):
    # Initially assigns one ticket to everyone.
    tickets = [1] * len(productivity)
    # From left to right.
    for i in range(1, len(productivity)):
        if productivity[i] > productivity[i - 1]:
            tickets[i] = tickets[i - 1] + 1
    # From right to left.
    for i in reversed(range(len(productivity) - 1)):
        if productivity[i] > productivity[i + 1]:
            tickets[i] = max(tickets[i], tickets[i + 1] + 1)
    return sum(tickets)
```

24.15 Search a sorted array of unknown length

Binary search is usually applied to an array of known length. Sometimes, the array is "virtual", i.e., it is an abstraction of data that is spread across multiple machines. In such cases, the length is not known in advance; accessing elements beyond the end results in an exception.

Design an algorithm that takes a sorted array whose length is not known, and a key, and returns an index of an array element which is equal to the key. Assume that an out-of-bounds access throws an exception.

Hint: Can divide and conquer be used to find the end of the array?

Solution: The brute-force approach is to iterate through the array, one element at a time, stopping when either the key is found or an exception is thrown (in which case the key is not present). The time complexity is $O(n)$, where n is the length of the input array.

A better approach is to take advantage of sortedness. If we know the array length, we can use binary search to search for the key. We can compute the array length by testing whether indices $0, 1, 3, 7, 15, \ldots$ are valid. As soon as we find an invalid index, say $2^i - 1$, we can use binary search over the interval $[2^{i-1}, 2^i - 2]$ to find the first invalid index, which is the length of the array.

We can improve on the above approach by comparing the value of the element at index $2^i - 1$ with the key, since if it is greater than the key, we can do binary search over indices $[2^{i-1}, 2^i - 2]$ for the key. Conceptually, we can treat out-of-bounds indices in the same way as valid indices by treating an out-of-bounds index as holding infinity.

For example, consider the array in Figure 11.1 on Page 145. Suppose we are searching for the key 243. We examine indices 0, 1, 3, 7. Since $A[7] = 285 > 243$, we now perform conventional binary search over the interval $[4, 6]$ for 243. If instead, we were searching for the key 400, we would examine indices $0, 1, 3, 7, 15$. Since 15 is not a valid index, we would stop, and perform conventional binary search over the interval $[8, 14]$. The first midpoint is 11, which is out-of-bounds and treated as holding infinity, so we update the interval to $[8, 10]$. The next interval is $[8, 8]$, followed by $[8, 7]$ which is empty, indicating that the key 400 is not present.

```
def binary_search_unknown_length(A, k):
    # Find the possible range where k exists.
    p = 0
    while True:
        try:
            idx = (1 << p) - 1  # 2^p - 1.
            if A[idx] == k:
                return idx
            elif A[idx] > k:
                break
        except IndexError:
            break
        p += 1

    # Binary search between indices 2^(p - 1) and 2^p - 2, inclusive.
    left, right = 1 << max(0, (p - 1)), (1 << p) - 2
    while left <= right:
        mid = left + (right - left) // 2
        try:
            if A[mid] == k:
                return mid
            elif A[mid] > k:
                right = mid - 1
            else:  # A[mid] < k
                left = mid + 1
        except IndexError:
```

```
            right = mid - 1  # Search the left part if out-of-bound.
    return -1  # Nothing matched k.
```

The run time of the first loop is $O(\log n)$, since we double the tested index with each iteration. The second loop is conventional binary search, i.e., $O(\log n)$, so the total time complexity is $O(\log n)$.

24.16 Search in two sorted arrays

You are given two sorted arrays and a positive integer k. Design an algorithm for computing the kth smallest element in an array consisting of the elements of the initial two arrays arranged in sorted order. Array elements may be duplicated within and across the input arrays.

Hint: The first k elements of the first array together with the first k elements of the second array are initial candidates. Iteratively eliminates a constant fraction of the candidates.

Solution: You could merge the two arrays into a third sorted array and then look for the answer—the merge would take $O(m+n)$ time, where m and n are the lengths of the input arrays.

You can optimize somewhat by building the merged array on the first k elements, which would be an $O(k)$ operation—this is faster than forming the combined array when k is small, but when k is comparable to m and n, the time complexity is $O(m+n)$.

What we really need is some form of binary search that takes advantage of the sortedness of A and B. Intuitively, if we focus on finding the indices in A and B that correspond to the first k elements, we stand a good chance of using binary search. Specifically, suppose the first k elements of the union of A and B consist of the first x elements of A and the first $k-x$ elements of B. We'll now see how to use binary search to determine x.

Let's maintain an interval $[b,t]$ that contains x. The iteration continues as long as $b < t$. We will contract this interval by half in each iteration. At each iteration consider the midpoint, $x = b + \lfloor \frac{t-b}{2} \rfloor$. If $A[x] < B[(k-x)-1]$, then $A[x]$ must be in the first $k-1$ elements of the union, so we update b to $x+1$ and continue. Similarly, if $A[x-1] > B[k-x]$, then $A[x-1]$ cannot be in the first k elements, so we can update t to $x-1$. Otherwise, we must have $B[(k-x)-1] \leq A[x]$ and $A[x-1] \leq B[k-x]$, in which case the result is the larger of $A[x-1]$ and $B[(k-x)-1]$, since the first x elements of A and the first $k-x$ elements of B when sorted end in $A[x-1]$ or $B[(k-x)-1]$.

If the iteration ends without returning, it must be that $b = t$. Clearly, $x = b = t$. We simply return the larger of $A[x-1]$ and $B[(k-x)-1]$. (If $A[x-1] = B[(k-x)-1]$, we arbitrarily return either.)

The initial values for b and t need to be chosen carefully. Naively setting $b = 0, t = k$ does not work, since this choice may lead to array indices in the search lying outside the range of valid indices. The indexing constraints for A and B can be resolved by initializing b to $\max(0, k-n)$ and t to $\min(m,k)$.

```
def find_kth_in_two_sorted_arrays(A, B, k):
    # Lower bound of elements we will choose in A.
    b = max(0, k - len(B))
    # Upper bound of elements we will choose in A.
    t = min(len(A), k)

    while b < t:
        x = b + (t - b) // 2
        A_x_1 = float('-inf') if x <= 0 else A[x - 1]
```

```
        A_x = float('inf') if x >= len(A) else A[x]
        B_k_x_1 = float('-inf') if k - x <= 0 else B[k - x - 1]
        B_k_x = float('inf') if k - x >= len(B) else B[k - x]

        if A_x < B_k_x_1:
            b = x + 1
        elif A_x_1 > B_k_x:
            t = x - 1
        else:
            # B[k - x - 1] <= A[x] and A[x - 1] < B[k - x].
            return max(A_x_1, B_k_x_1)

    A_b_1 = float('-inf') if b <= 0 else A[b - 1]
    B_k_b_1 = float('-inf') if k - b - 1 < 0 else B[k - b - 1]
    return max(A_b_1, B_k_b_1)
```

Since in each iteration we halve the length of $[b, t]$ the time complexity is $O(\log k)$.

24.17 Find the kth largest element—large n, small k

The goal of this problem is to design an algorithm for computing the kth largest element in a sequence of elements that is presented one element at a time. The length of the sequence is not known in advance, and could be very large.

Design an algorithm for computing the kth largest element in a sequence of elements.

Hint: Track the k largest elements, but don't update the collection immediately after each new element is read.

Solution: The natural approach is to use a min-heap containing the k largest elements seen thus far, just as in Solution 10.4 on Page 138. When the last element in the sequence is read, the desired value is the element at the root of the min-heap. This approach has time complexity $O(n \log k)$, where n is the total number of elements in the sequence.

We know of a very fast algorithm for finding the kth largest element in an array of fixed size (Solution 11.8 on Page 153). We cannot directly apply that here without allocating $O(n)$ space. However, we can break our input into fixed size arrays and run Solution 11.8 on Page 153 over those arrays to eliminate all but the k largest elements from each array. These elements are added over to the next array.

```
def find_kth_largest_unknown_length(stream, k):
    candidates = []
    for x in stream:
        candidates.append(x)
        if len(candidates) >= 2 * k - 1:
            # Reorders elements about median with larger elements appearing
            # before the median.
            find_kth_largest(k, candidates)
            # Resize to keep just the k largest elements seen so far.
            del candidates[k:]
    # Finds the k-th largest element in candidates.
    find_kth_largest(k, candidates)
    return candidates[k - 1]
```

By using $2k-1$ as the array size, the time complexity to find the kth largest element is almost certain $O(k)$. It is run every $k-1$ elements, implying an $O(n)$ time complexity.

Note that we could use less storage, e.g., an array of length $3k/2$, and still achieve $O(n)$ time complexity. The actual run time would be higher with an array of length $3k/2$ since we only discard $k/2$ elements for each call to finding the kth largest element. This is a classic space-time trade-off. If we used a $4k$ long array, we could discard $3k$ elements for one call to Solution 11.8 on Page 153. The time complexity of Solution 11.8 on Page 153 is proportional to the length of the array, so using a length $4k$ array compared to a length $3k/2$ array yields a speed-up of $\frac{(3/2)}{(1/2)/(4/3)} = 2.25$. Clearly more storage leads to faster run times (in the extreme we read all n and do a single call to Solution 11.8 on Page 153), so there is a trade-off with respect to how much storage we want.

24.18 Find an element that appears only once

Given an integer array of length n, where each element except for one appears twice, with the remaining element appearing only once, we can use $O(n)$ space and $O(n)$ time to find the element that appears exactly once, e.g., using a hash table. However, there is a better solution: compute the bitwise-XOR ($\oplus$) of all the elements in the array. Because $x \oplus x = 0$, all elements that appear an even number of times cancel out, and the element that appears exactly once remains. Therefore, this problem can be solved using $O(1)$ space.

Given an integer array, in which each entry but one appears in triplicate, with the remaining element appearing once, find the element appearing once. For example, if the array is $\langle 2,4,2,5,2,5,5\rangle$, you should return 4.

Hint: Count the number of 1s at each index.

Solution: The brute-force solutions in Solution 24.18, namely using a hash table or sorting will work for this problem too, with the same time and space complexities.

One way to view Solution 11.10 on Page 157 is that it counts mod 2 for each bit-position the number of entries in which the bit in that position is 1. Specifically, the XOR of elements at indices $[0, i-1]$, determines exactly which bit-positions have been odd number of times in elements of the input array whose indices are in $[0, i-1]$.

The analogous approach for the current problem is to count mod 3 for each bit-position the number of times the bit in that position has been 1. The effect of counting mod 3 is to eliminate the elements that appear three times, and so the bit-positions which have a count of 1 are precisely those bit-positions in the count which are set to 1.

The example array, $\langle 2,4,2,5,2,5,5\rangle$, expressed in binary is $\langle (010)_2, (100)_2, (010)_2, (101)_2, (010)_2, (101)_2, (101)_2\rangle$. The number of bits set to 1 in position 0 (the LSB) across all 7 array entries is 3; the number of bits set to 1 in position 1 is 3, and the number of bits set to 1 in position 2 is 4. By taking each of these quantities mod 3 we cast out the contributions of elements that appear exactly three times, which leaves us with a 1 in the MSB and 0 in the remaining two positions, i.e., the element which only appeared once.

We can implement the above idea using an array C of integers whose length equals the integer word size. Entry $C[i]$ will be used to count the number of 1s in bit-position i, across all the inputs.

By the above argument, after the entire input is processed, $C[i]$ mod 3 will be 1 at exactly those bit-positions where the input that appears once has a 1.

```
def find_element_appears_once(A):
    counts = [0] * 32
    for x in A:
        for i in range(32):
            counts[i] += x & 1
            x >>= 1

    def handle_negative(n):
        return n if n < 2**31 else n - 2**32

    # Any result greater than or equal to 2**31 must correspond to an negative
    # value.
    return handle_negative(sum(1 << i for i, c in enumerate(counts) if c % 3))
```

The time complexity is $O(n)$ and space complexity is $O(1)$.

Variant: Solve the same problem when one entry appears twice and the rest appear three times.

24.19 Find the line through the most points

You are given a set of points in the plane. Each point has integer coordinates. Design an algorithm for computing a line that contains the maximum number of points in the set.

Hint: A line can be uniquely represented by two numbers.

Solution: This problem may seem daunting at first—there are literally infinitely many lines. The only lines we care about are those that pass through points in the set, and, more specifically, lines that pass through at least two points in the set.

A brute-force approach then is to compute all such lines, and for each such line, count exactly how many points from the set lie on it. We can use a hash table to represent the set of lines. The set of points corresponding to a line could itself be stored using a hash table. If there are n points in the set, there are $n(n-1)/2$ pairs of points. Naively, we would compute the set of lines defined by pairs of points, and then iterate over all points, checking for each line if that point belongs to that line. The time complexity is dominated by the iteration over points and lines, which is $O(n \times n(n-1)/2) = O(n^3)$.

A better approach is to add the pair of points to the set of points on the line they define immediately. for each pair we have to do a lookup, an insert into the hash table if the defined line is not already present, and two inserts into the corresponding set of points. The hash table operations are $O(1)$ time, leading to an $O(n^2)$ time complexity for this part of the computation. The space complexity is also $O(n^2)$. This is a consequence of the time complexity. At a first glance, it seems like the space complexity might be higher, since there are $O(n^2)$ pairs of lines, and each can have up to $O(n)$ points. However, there is an inverse relationship between the number of lines and the number of points per line.

We finish by finding the line with the maximum number of points with a simple iteration through the hash table searching for the line with the most points in its corresponding set. There are at most $n(n-1)/2$ lines, so the iteration takes $O(n^2)$ time, yielding an overall time bound of $O(n^2)$.

The design of a hash function appropriate for lines is more challenging than it may seem at first. The equation of the line through (x_1, y_1) and (x_2, y_2) is

$$y = \frac{y_2 - y_1}{x_2 - x_1}x + \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1}.$$

One idea would be to compute a hash code from the slope and the Y-intercept of this line as an ordered pair of floating-point numbers. However, because of finite precision arithmetic, we may have three points that are collinear map to distinct buckets.

A more robust hash function treats the slope and the Y-intercept as rationals. A rational is an ordered pair of integers: the numerator and the denominator. We need to bring the rational into a canonical form before applying the hash function. One canonical form is to make the denominator always nonnegative, and relatively prime to the numerator. Lines parallel to the Y-axis are a special case. For such lines, we use the X-intercept in place of the Y-intercept, and use $\frac{1}{0}$ as the slope.

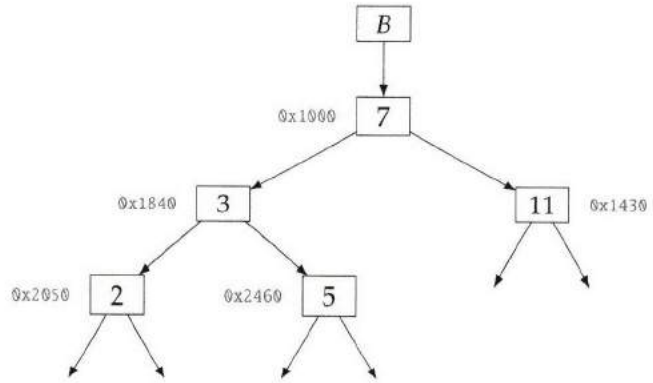
```
Point = collections.namedtuple("Point", ("x", "y"))


def find_line_with_most_points(points):
    result = 0
    for i, p1 in enumerate(points):
        slope_table = collections.defaultdict(int)
        overlap_points = 1
        for p2 in points[i + 1:]:
            if p1 == p2:
                overlap_points += 1
            elif p1.x == p2.x:
                # A vertical line with slope 1/0.
                slope_table[(0, 1)] += 1
            else:
                x_diff, y_diff = p1.x - p2.x, p1.y - p2.y
                gcd = fractions.gcd(x_diff, y_diff)
                x_diff, y_diff = x_diff / gcd, y_diff / gcd
                slope_table[(x_diff, y_diff)] += 1
        result = max(result,
                     overlap_points + max(slope_table.values(), default=0))
    return result
```

24.20 Convert a sorted doubly linked list into a BST

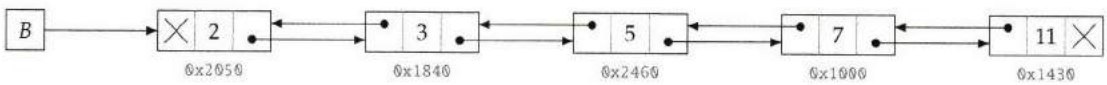
Lists and BSTs are both examples of "linked" data structures, i.e., some fields are references to other objects of the same type. Since nodes in a doubly linked list and in a BST both have a key field and two references, it's natural to consider the following problem.

Write a program that takes as input a doubly linked list of sorted numbers and builds a height-balanced BST on the entries in the list. Reuse the nodes of the list for the BST, using the previous and next fields for the left and right children, respectively. See Figure 24.6(b) on the next page for an example of a doubly linked list, and Figure 24.6(a) on the facing page for the BST on the same nodes.

Hint: Update reference fields, not node contents.



(a) A BST on five nodes—edges that do not terminate in nodes denote empty subtrees. The number in hex adjacent to each node represents its address in memory.



(b) The sorted doubly linked list corresponding to the BST in (a). Note how the tree nodes have been used for the list nodes.

Figure 24.6: BST and sorted doubly linked list interconversion.

Solution: If the list nodes were in an array, we could index directly into the array to obtain the midpoint, and the time complexity would satisfy $T(n) = O(1) + 2T(\frac{n}{2})$, where n is the number of nodes in the list, which solves to $T(n) = O(n)$. This is the approach of Solution 14.8 on Page 210. We can recycle the list nodes, but creating the array entails $O(n)$ additional space.

A direct approach to the construction which does not allocate new nodes is to find the midpoint of the list, and use it as the root, recursing on the first half and the second half of the list. The time complexity satisfies the recurrence $T(n) = O(n) + 2T(\frac{n}{2})$—the $O(n)$ term comes from the traversal required to find the midpoint of the list, which itself entails computing the length of the list. This solves to $T(n) = O(n \log n)$. The added time complexity compared to the array-based approach comes from the inability to find a midpoint in a list in $O(1)$ time.

The key insight to improving the time complexity without adding to the space complexity is noting that since we have to spend $O(n)$ time to find the midpoint, we should do more than just get the midpoint. Specifically, we can first create a balanced BST on the first $\lfloor \frac{n}{2} \rfloor$ nodes. Then we use the $(\lfloor \frac{n}{2} \rfloor + 1)$th node as the root of the final BST and set its left child to the BST just created.

Since we are changing the links in the list, we need to be careful to ensure we can recover the root. We can do this by keeping a reference to the head of the list being processed, and advancing this reference inside the recursive calls. This allows us to compute the root while computing the left subtree.

Finally we create a balanced BST on the remaining $n - \lfloor \frac{n}{2} \rfloor - 1$ nodes, and set it as the root's right child.

```
# Returns the root of the corresponding BST. The prev and next fields of the
# list nodes are used as the BST nodes left and right fields, respectively.
# The length of the list is given.
def build_bst_from_sorted_doubly_list(L, n):
    # Builds a BST from the (start + 1)-th to the end-th node, inclusive, in L,
    # and returns the root.
```

```
    def build_bst_from_sorted_doubly_list_helper(start, end):
        if start >= end:
            return None

        mid = (start + end) // 2
        left = build_bst_from_sorted_doubly_list_helper(start, mid)
        # The last function call sets L to the successor of the maximum node in
        # the tree rooted at left.
        curr, head[0] = head[0], head[0].next
        curr.prev = left
        curr.next = build_bst_from_sorted_doubly_list_helper(mid + 1, end)
        return curr

    head = [L]
    return build_bst_from_sorted_doubly_list_helper(0, n)
```

The algorithms spends $O(1)$ time per node, leading to an $O(n)$ time complexity. No dynamic memory allocation is required. The maximum number of call frames in the function call stack is $\lceil \log n \rceil$, yielding an $O(\log n)$ space complexity.

24.21 Convert a BST to a sorted doubly linked list

A BST node has two references, left and right. A doubly linked list node has two references, previous and next. If we interpret the BST's left pointer as previous and the BST's right pointer as next, a BST's node can be used as a node in a doubly linked list. Also, the inorder traversal of a BST represents an ordered set just like a doubly linked list. Therefore it is natural to ask if is possible to take a BST and rewrite its node reference fields so that it represents a doubly linked list such that the resulting list represents the inorder traversal sequence of the tree.

Design an algorithm that takes as input a BST and returns a sorted doubly linked list on the same elements. Your algorithm should not do any dynamic allocation. The original BST does not have to be preserved; use its nodes as the nodes of the resulting list, as shown in Figure 24.6 on the preceding page.

Hint: The tricky part is attaching the root to its subtrees.

Solution: In the absence of the allocation constraint, the problem can be easily solved using a dynamic array to write nodes to a list as we perform an inorder traversal. The time complexity is $O(n)$, where n is the number of nodes, but the space complexity is also $O(n)$.

Speaking generally, a key benefit of lists is that we can easily append one list to another. In particular, if we have lists for the left and right subtrees, we can easily splice them in with the root in $O(1)$ time.

```
def bst_to_doubly_linked_list(tree):
    HeadAndTail = collections.namedtuple('HeadAndTail', ('head', 'tail'))

    # Transforms a BST into a sorted doubly linked list in-place,
    # and return the head and tail of the list.
    def bst_to_doubly_linked_list_helper(tree):
        # Empty subtree.
        if not tree:
```

```
            return HeadAndTail(None, None)

        # Recursively builds the list from left and right subtrees.
        left = bst_to_doubly_linked_list_helper(tree.left)
        right = bst_to_doubly_linked_list_helper(tree.right)

        # Appends tree to the list from left subtree.
        if left.tail:
            left.tail.right = tree
        tree.left = left.tail

        # Appends the list from right subtree to tree.
        tree.right = right.head
        if right.head:
            right.head.left = tree

        return HeadAndTail(left.head or tree, right.tail or tree)

    return bst_to_doubly_linked_list_helper(tree).head
```

Since we do a constant amount of work per tree node, the time complexity is $O(n)$. The space complexity is the maximum depth of the function call stack, i.e., $O(h)$, where h is the height of the BST. The worst-case is for a skewed tree—n activation records are pushed on the stack.

24.22 Merge two BSTs

Given two BSTs, it is straightforward to create a BST containing the union of their keys: traverse one, and insert its keys into the other.

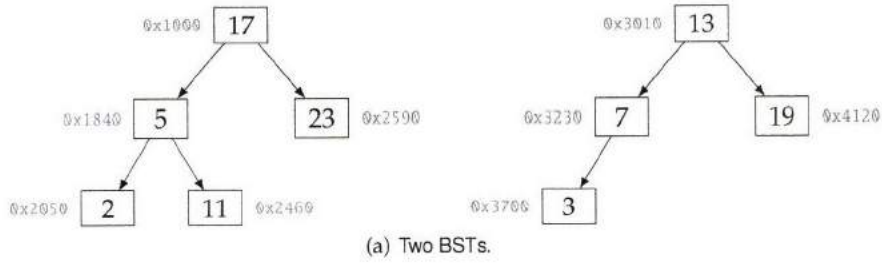
Design an algorithm that takes as input two BSTs and merges them to form a balanced BST. For any node, you can update its left and right subtree fields, but cannot change its key. See Figure 24.7 on the next page for an example. Your solution can dynamically allocate no more than a few bytes.

Hint: Can you relate this problem to Problems 24.21 on the facing page and 24.20 on Page 376?

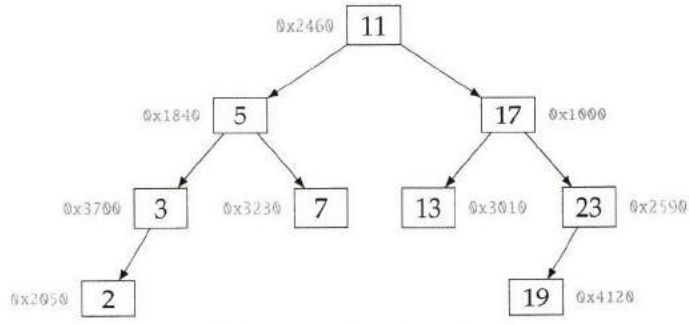
Solution: A brute-force approach is to traverse one tree and add its keys to the second tree. The time complexity depends on how balanced the second tree is, and how we perform the insert. In the best-case, we start with the second tree being balanced, and preserve balance as we perform additions, yielding a time complexity of $O(n \log n)$, where n is the total number of nodes. Performing the updates while reusing existing nodes is tricky if we do an inorder walk since the links change. However, it is fairly simple if we do a post-order walk, since when we visit the node, we do not need any information from its original left and right subtrees.

Looking more carefully at the brute-force approach, it is apparent that it does not exploit the fact that both the sources of data being merged are sorted. In particular, if memory was not a constraint, we could perform an inorder walk on each tree, writing the result of each to a sorted array. Then we could put the union of the two arrays into a third sorted array. Finally, we could build a balanced BST from the union array using recursion, e.g., using Solution 14.8 on Page 210. The time complexity would be $O(n)$, but the additional space complexity is $O(n)$.

It is good to remember that a list can be viewed as a tree in which each node's left child is empty. It is relatively simple to create a list of the same nodes as a tree (Solution 24.21 on Page 378). We



(a) Two BSTs.



(b) Merged BST corresponding to the two BSTs in (a).

Figure 24.7: Example of merging two BSTs. The number in hex adjacent to each node represents its address in memory.

can take these two lists and form a new list on the same nodes which is the union of the keys in sorted order (Solution 7.1 on Page 84). This new list can be viewed as a tree holding the union of the keys of the original trees. The time complexity is $O(n)$, and space complexity is $O(h)$, where h is the maximum of the heights of the two initial trees.

The problem with this approach is that while it meets the time and space constraints, it returns a tree that is completely unbalanced. However, we can convert this tree to a height-balanced one using Solution 24.20 on Page 377, completing the desired construction.

```
def merge_two_bsts(A, B):
    A, B = bst_to_doubly_list(A), bst_to_doubly_list(B)
    A_length, B_length = count_length(A), count_length(B)
    return build_bst_from_sorted_doubly_list(
        merge_two_sorted_lists(A, B), A_length + B_length)
```

The time complexity of each stage is $O(n)$, and since we recycle storage, the additional space complexity is dominated by the time to convert a BST to a list, which is $O(h)$.

24.23 Implement regular expression matching

A regular expression is a sequence of characters that forms a search pattern. For this problem we define a simple subset of a full regular expression language. We describe regular expressions by examples, rather than a formal syntax and semantics.

A regular expression is a string made up of the following characters: alphanumeric, . (dot), * (star), ^, and $. Examples of regular expressions are a, aW, aW.9, aW.9*, aW.*9*, ^a, aW$, and ^aW.9*$. Not all strings are valid regular expressions. For example, if ^ appears, it must be the first

character, if $ appears, it must be the last character, and star must follow an alphanumeric character or dot. Beyond the base cases—a single alphanumeric character, dot, a single alphanumeric character followed by a star, dot followed by star—regular expressions are concatenations of shorter regular expressions.

Now we describe what it means for a regular expression to match a string. Intuitively, an alphanumeric character matches itself, dot matches any single character, and star matches zero or more occurrences of the preceding character.

In the absence of ^ and $, there is no concept of an "anchor". In particular, if the string contains any substring matched by the regular expression, the regular expression matches the string itself.

The following examples illustrate the concept of a regular expression matching a string. More than one substring may be matched. If a match exists, we underline a matched substring.

- aW9 matches any string containing aW9 as a substring. For example, aW9, aW9bcW, ab8aW9, and cc2aW9raW9z are all matched by aW9, but aW8, bcd8, and xy are not.
- a.9. matches any string containing a substring of length 4 whose first and third characters are a and 9, respectively. For example, ab9w, ac9bcW, ab8a999, and cc2aW9r are all matched by a.9., but az9, a989a, and bac9 are not.
- aW*9 matches any string containing a substring beginning with a, ending with 9, with zero or more Ws in between. For example, a9, aW9, aWW9b9cW, aU9aWW9, ab8aWWW9W9aa, and cc2a9raW9zWW9ac are all matched by aW*9, but aWWU9, baX9, and aXW9Wa are not.
- a.*9 matches any string containing a substring beginning with a ending with 9, with zero or more characters between. For example, a9, aZ9, aZW9b9cW, aU9a9, b8aWUW9W, and cc2a9raU9z are all matched by a.*9, but 9UWaW8, b9aaaX, and XUq8 are not.
- aW9*.b3 matches any string containing a substring beginning with aW, followed by zero or more 9s, followed by a single character, followed by b3. For example, ceaW999zb34b3az, ceaW9b34, and pqaWzb38q are matched by aW9*.b3, but ceaW98zb34 and pqaW988b38q are not.

If the regular expression begins with ^, that indicates the match must begin at the start of the string. If the regular expression ends with $, the match must end at the end of the string.

- ^aW.9 matches strings which begin with a substring consisting of a, followed by W, followed by any character, followed by 9. For example, aW99zer, aWW9, and aWP9GA are all matched by ^aW.9, but baWx9, aW9, and aWcc90 are not.
- aW.9$ matches strings whose last character is 9, third last character is W and fourth last character is a. (The second last character can be anything.) For example, aWW9, aWW9abcaWz9, baaWX9, and abcaWP9, are all matched by aW.9$, but aWW99, aW, and aWcc90 are not.
- ^aW9$ is matched by aW9 and nothing else.

Design an algorithm that takes a regular expression and a string, and checks if the regular expression matches the string.

Hint: Regular expressions are defined recursively.

Solution: The key insight is that regular expressions are defined recursively, both in terms of their syntax (what strings are valid regular expressions), as well as their semantics (when does a regular expression match a string). This suggests the use of recursion to do matching.

First, some notation: s^k denotes the kth suffix of string s, i.e., the string resulting from deleting the first k characters from s. For example, if s = aWaW9W9, then s^0 = aWaW9W9, and s^2 = aW9W9.

Let r be a regular expression and s a string. If r starts with ^, then for r to match s, the remainder of r must match a prefix of s. If r ends with a \$, then for r to match s, some suffix of s must be matched by r without the trailing \$. If r does not begin with ^, or end with \$, r matches s if it matches some substring of s.

A function that checks whether a regular expression matches a string at its beginning has to check the following cases:

(1.) Length-0 regular expressions.

(2.) A regular expression starting with ^ or ending with \$.

(3.) A regular expression starting with a * match, e.g., a*wXY or .*Wa.

(4.) A regular expression starting with an alphanumeric character or dot.

Case (1.) is trivial, we just return true. Case (2.) entails a single call to the match function for a regular expression beginning with ^, and some checking logic for a regular expression ending with \$. Case (3.) is handled by a traversal down the string checking that the prefix of the string thus far matches the alphanumeric character or dot until some suffix is matched by the remainder of the regular expression. Each suffix check is a call to the match function. Case (4.) involves examining a character, possibly followed by a call to the match function.

As an example, consider the regular expression ab.c*d. To check if it matches s = caeabbedeabacccde, we iterate over the string. Since s[0] = c, we cannot match the regular expression at the start of s. Next we try s^1. Since s[1] = a, we recursively continue checking with s^1 and b.c*d. However s[2] $\neq$ b, so we return false from this call. Continuing, s^2 is immediately eliminated. With s^3, since s[3] = a, we continue recursively checking with s^4 and b.c*d. Since s[4] = b, we continue checking with .c*d. Since dot matches any single character, it matches s[5], so we continue checking with c*d. Since s[6] = e, the only prefix of s^6 which matches c*d is the empty one. However, when we continue checking with d, since s[6] $\neq$ d, we return false for this call. Skipping some unsuccessful checks, we get to s^9. Similar to before, we continue to s^{12}, and c*d. The string beginning at offset 12 matches c* with prefixes of length 0, 1, 2, 3. After the first three, we do not match with the remaining d. However, after the prefix ccc, the following string does end in d, so we return true.

```
def is_match(regex, s):
    def is_match_here(regex, s):
        if not regex:
            # Case (1.): Empty regex matches all strings.
            return True

        if regex == '$':
            # Case (2.): Reach the end of regex, and last char is '$'.
            return not s

        if len(regex) >= 2 and regex[1] == '*':
            # Case (3.): A '*' match.
            # Iterate through s, checking '*' condition, if '*' condition holds,
            # performs the remaining checks.
            i = 1
            while i <= len(s) and regex[0] in ('.', s[i - 1]):
                if is_match_here(regex[2:], s[i:]):
                    return True
                i += 1
```

```
        # See '*' matches zero character in s[:len(s)].
        return is_match_here(regex[2:], s)

    # Case (4.): regex begins with single character match.
    return bool(s and regex[0] in ('.', s[0])
                and is_match_here(regex[1:], s[1:]))

# Case (2.): regex starts with '^'.
if regex[0] == '^':
    return is_match_here(regex[1:], s)
return any(is_match_here(regex, s[i:]) for i in range(len(s) + 1))
```

Let $C(x,k)$ be k copies of the string x concatenated together. For the regular expression $C(a.*,k)$ and string $C(ab,k-1)$ the algorithm presented above takes time exponential in k. We cannot give a more precise bound on the time complexity.

Variant: Solve the same problem for regular expressions without the ^ and $ operators.

24.24 Synthesize an expression

Consider an expression of the form $\langle 3 \odot 1 \odot 4 \odot 1 \odot 5 \rangle$, where each $\odot$ is an operator, e.g., $+, -, \times, \div$. The expression takes different values based on what the operators are. Some examples are 14 (if all operators are +), 60 (if all operators are $\times$), and 22 ($3 - 1 + 4 \times 1 \times 5$).

Determining an operator assignment such that the resulting expression takes a specified value is, in general, a difficult computational problem. For example, suppose the operators are + and −, and we want to know whether we can select each $\odot$ such that the resulting expression evaluates to 0. The problem of partitioning a set of integers into two subsets which sum up to the same value, which is a famous NP-complete problem, directly reduces to our problem.

Write a program that takes an array of digits and a target value, and returns true if it is possible to intersperse multiplies ($\times$) and adds (+) with the digits of the array such that the resulting expression evaluates to the target value. For example, if the array is $\langle 1,2,3,2,5,3,7,8,5,9 \rangle$ and the target value is 995, then the target value can be realized by the expression $123 + 2 + 5 \times 3 \times 7 + 85 \times 9$, so your program should return true.

Hint: Build the assignment incrementally.

Solution: Let A be the array of digits and k the target value. We want to intersperse $\times$ and + operations among these digits in such a way that the resulting expression equals k.

For each pair of characters, $(A[i], A[i+1])$, we can choose to insert a $\times$, a +, or no operator. If the length of A is n, the number of such locations is $n-1$, implying we can encode the choice with an array of length $n-1$. Each entry is one of three values—$\times$, +, and $\sqcup$ (which indicates no operator is added at that location). There are exactly 3^{n-1} such arrays, so a brute-force solution is to systematically enumerate all arrays. For each enumerated array, we compute the resulting expression, and return as soon as we evaluate to k. The time complexity is $O(n \times 3^n)$, since each expression takes time $O(n)$ to evaluate.

To improve runtime, we use a more focused enumeration. Specifically, the first operator can appear after the first, second, third, etc. digit, and it can be a + or $\times$. For + to be a possibility, it must

be that the sum of the value given by the initial operator assignment plus the value encoded by the remaining digits is greater than or equal to the target value. This is because the maximum value is achieved when there are no operators. For example, for $\langle 1,2,3,4,5\rangle$, we can never achieve a target value of 1107 if the first operator is a + placed after the 3 (since $123 + 45 < 1107$). This gives us a heuristic for pruning the search.

```
def expression_synthesis(digits, target):
    def directed_expression_synthesis(digits, current_term):
        def evaluate():
            intermediate_operands = []
            operand_it = iter(operands)
            intermediate_operands.append(next(operand_it))
            # Evaluates '*' first.
            for oper in operators:
                if oper == '*':
                    product = intermediate_operands[-1] * next(operand_it)
                    intermediate_operands[-1] = product
                else:  # oper == '+'.
                    intermediate_operands.append(next(operand_it))
            # Evaluates '+' second.
            return sum(intermediate_operands)

        current_term = current_term * 10 + digits[0]
        if len(digits) == 1:
            operands.append(current_term)
            if evaluate() == target:  # Found a match.
                return True
            del operands[-1]
            return False

        # No operator.
        if directed_expression_synthesis(digits[1:], current_term):
            return True
        # Tries multiplication operator '*'.
        operands.append(current_term)
        operators.append('*')
        if directed_expression_synthesis(digits[1:], 0):
            return True
        del operands[-1]
        del operators[-1]
        # Tries addition operator '+'.
        operands.append(current_term)
        # First check feasibility of plus operator.
        if target - evaluate() <= functools.reduce(lambda val, d: val * 10 + d,
                                                   digits[1:], 0):
            operators.append('+')
            if directed_expression_synthesis(digits[1:], 0):
                return True
            del operators[-1]
        del operands[-1]
        return False

    operands, operators = [], []
    return directed_expression_synthesis(digits, 0)
```

Despite the heuristics helping in some cases, we cannot prove a better bound for the worst-case time complexity than the original $O(n3^n)$.

24.25 Count inversions

Let A be an array of integers. Call the pair of indices (i, j) inverted if $i < j$ and $A[i] > A[j]$. For example, if $A = \langle 4, 1, 2, 3 \rangle$, then the pair of indices $(0, 3)$ is inverted. Intuitively, the number of inverted pairs in an array is a measure of how unsorted it is.

Design an efficient algorithm that takes an array of integers and returns the number of inverted pairs of indices.

Hint: Let A and B be arrays. How would you count the number of inversions where one element is from A and the other from B in the array consisting of elements from A and B?

Solution: The brute-force algorithm examines all pairs of indices (i, j), where $i < j$. has an $O(n^2)$ complexity, where n is the length of the array.

One way to recognize that the brute-force algorithm is inefficient is to consider how we would check if an array is sorted. We would not test for every element if all subsequent elements are greater than or equal to it—we just test the next element, since greater than or equal is transitive.

This suggests the use of sorting to speed up counting the number of inverted pairs. In particular, if we sort the second half of the array, then to see how many inversions exist with an element in the first half, we could do a binary search for that element in the second half. The elements before its location in the second half are the ones inverted with respect to it.

Elaborating, suppose we have counted the number of inversions in the left half L and the right half R of A. What are the inversions that remain to be counted? Sorting L and R makes it possible to efficiently obtain this number. For any (i, j) pair where i is an index in L and j is an index in R, if $L[i] > R[j]$, then for all $j' < j$ we must have $L[i] > R[j']$.

For example, if $A = \langle 3, 6, 4, 2, 5, 1 \rangle$, then $L = \langle 3, 6, 4 \rangle$, and $R = \langle 2, 5, 1 \rangle$. After sorting, $L = \langle 3, 4, 6 \rangle$, and $R = \langle 1, 2, 5 \rangle$. The inversion counts for L and R are 1 and 2, respectively. When merging to form their sorted union, since $1 < 3$, we know 4, 6 are also inverted with respect to 1, so we add $|L| - 0 = 3$ to the inversion count. Next we process 2. Since $2 < 3$, we know 4, 6 are also inverted with respect to 2, so we add $|L| - 0 = 3$ to the inversion count. Next we process 3. Since $5 > 3$, 3 does not add any more inversions. Next we process 4. Since $5 > 4$, 4 does not add any more inversions. Next we process 5. Since $5 < 6$, we add $|L| - 2$ (which is the index of 6) $= 1$ to the inversion count. In all we add $3 + 3 + 1 = 7$ to inversion counts for L and R (which were 1 and 2, respectively) to get the total number of inversions, 10.

```
def count_inversions(A):
    # Return the number of inversions in A[start:end].
    def count_subarray_inversions(start, end):
        # Merge two sorted subarrays A[start:mid] and A[mid:end] into
        # A[start:end] and return the number of inversions across A[start:mid]
        # and A[mid:end].
        def merge_sort_and_count_inversions_across_subarrays(start, mid, end):
            sorted_A = []
            left_start, right_start, inversion_count = start, mid, 0
```

```
        while left_start < mid and right_start < end:
            if A[left_start] <= A[right_start]:
                sorted_A.append(A[left_start])
                left_start += 1
            else:
                # A[left_start:mid] are the inversions of A[right_start].
                inversion_count += mid - left_start
                sorted_A.append(A[right_start])
                right_start += 1

        # Updates A with sorted_A.
        A[start:end] = sorted_A + A[left_start:mid] + A[right_start:end]
        return inversion_count

    if end - start <= 1:
        return 0
    mid = (start + end) // 2
    return (
        count_subarray_inversions(start, mid) + count_subarray_inversions(
            mid, end) + merge_sort_and_count_inversions_across_subarrays(
                start, mid, end))

return count_subarray_inversions(0, len(A))
```

The time complexity satisfies $T(n) = O(n) + 2T(n-1)$, which solves to $O(n \log n)$, where n is the length of the array.

Variant: Runners numbered from 0 to $n-1$ race on a straight one-way road to a common finish line. The runners have different (constant) speeds and start at different distances from the finish line. Specifically, Runner i has a speed s_i and begins at a distance d_i from the finish line. Each runner stops at the finish line, and the race ends when all runners have reached the finish line. How many times does one runner pass another?

24.26 Draw the skyline

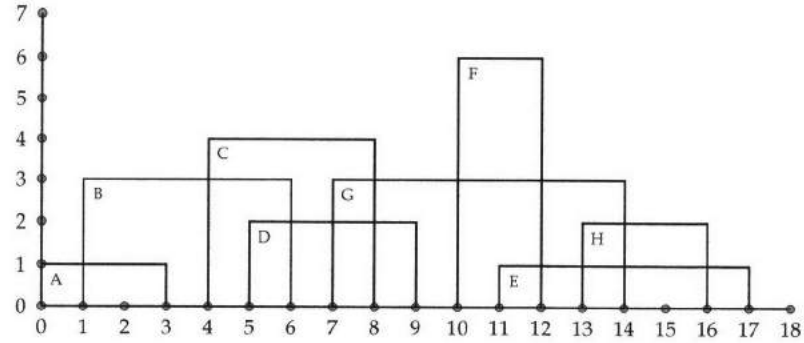
A number of buildings are visible from a point. A building appears as a rectangle, with the bottom of each building lying on a fixed horizontal line. A building is specified using its left and right coordinates, and its height. One building may partly obstruct another, as shown in Figure 24.8(a) on the facing page. The skyline is the list of coordinates and corresponding heights of what is visible.

For example, the skyline corresponding to the buildings in Figure 24.8(a) on the next page is given in Figure 24.8(b) on the facing page. (The patterned rectangles within the skyline are used to describe Problem 17.8 on Page 270; they are not relevant to the current problem.)

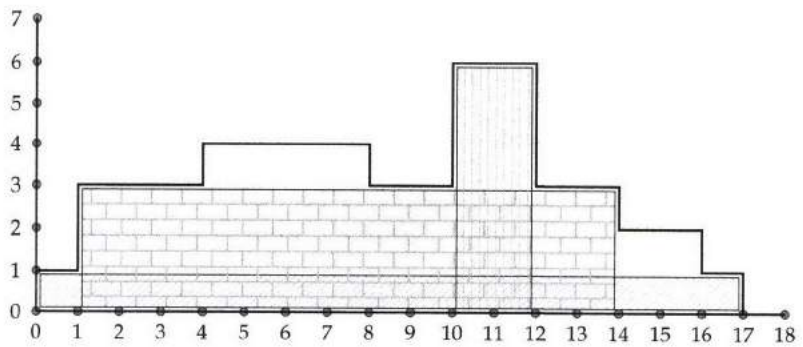
Design an efficient algorithm for computing the skyline.

Hint: Think of an efficient way of merging skylines.

Solution: The simplest solution is to compute the skyline incrementally. For one building, the skyline is trivial. Suppose we know the skyline for some buildings, and need to compute the skyline when another building is added. Let the new building's left and right coordinates be L and R, and its height H. To add it, we iterate through the existing skyline from left to right to see where



(a) A set of buildings.



(b) Skyline for the buildings in (a).

Figure 24.8: Buildings, their skyline, and the largest contained rectangle. The text label identifying the building is just below and to the right of its upper left-hand corner.

L should be added. Then we move through the existing skyline until we pass R, increasing any heights that are less than H to H.

This algorithm is simple, but has $O(n^2)$ complexity if there are n buildings, since adding the nth building may entail $O(n)$ comparisons. The key to improving efficiency is the observation that it takes linear time to merge two skylines (if they are represented in left-to-right order), which is the same as the time to merge a single skyline, but gets much more done.

Now it is clear what the textbook solution is: use divide-and-conquer to compute skylines for one half of the buildings, the other half of the buildings, and then merge the results. The merge is similar to the procedure for adding a single building, described above, and can be performed in $O(n)$ time. We iterate through the two skylines together from left-to-right, matching their X coordinates, and updating heights appropriately. Our website has a link to a program based on the algorithm above. The time complexity $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(n)$, where the latter term comes from the merge step. This solves to $T(n) = O(n \log n)$.

As an example, consider merging the skyline for Buildings A, B, C, D with the skyline for Buildings E, F, G, H. From 0 to 7, the skyline for A, B, C, D determines the height, because these are the only buildings present. At 7, since the skyline for A, B, C, D is taller, than that for E, F, G, H, we use its height, 4, going forward. At 8, since the skyline for E, F, G, H is taller, so we use its height, 3, going forward. After 9, the skyline for E, F, G, H determines the height, because these are the only buildings going forward.

Here is an alternative to the textbook solution that is hugely simpler, and, except for degenerate inputs, performs much better. It is based on "digitizing" the problem. Let's say the left-most coordinate for any building is l and the right-most coordinate for any building is r. Then the final skyline will start at l and end at r. We draw the the skyline building by building as follows. We represent the skyline with an array, where the ith element of the array hold the height of coordinate i of the current skyline. This array is initialized to 0s. For each building, we iterate over the coordinates, and update the skyline—if the current skyline's height at coordinate i is less than the height of the building, we update the skyline at i to the building's height.

```
Rectangle = collections.namedtuple('Rectangle', ('left', 'right', 'height'))


def compute_skyline(buildings):
    min_left = min(buildings, key=lambda b: b.left).left
    max_right = max(buildings, key=lambda b: b.right).right

    heights = [0] * (max_right - min_left + 1)
    for building in buildings:
        for i in range(building.left, building.right + 1):
            heights[i - min_left] = max(heights[i - min_left], building.height)

    result = []
    left = 0
    for i in range(1, len(heights)):
        if heights[i] != heights[i - 1]:
            result.append(
                Rectangle(left + min_left, i - 1 + min_left, heights[i - 1]))
            left = i
    return result + [Rectangle(left + min_left, max_right, heights[-1])]
```

The time complexity is $O(nW)$, where W is the width of the widest building. In theory, W could be very large, making this approach much worse than the textbook solution. In practice, W will be a constant, and the digitized solution will be much faster. It is also vastly simpler to code and to understand.

Variant: Solve the skyline problem when each building has the shape of an isosceles triangle with a 90 degree angle at its apex.

24.27 Measure with defective jugs

You have three measuring jugs, A, B, and C. The measuring marks have worn out, making it impossible to measure exact volumes. Specifically, each time you measure with A, all you can be sure of is that you have a volume that is in the range $[230, 240]$ mL. (The next time you use A, you may get a different volume—all that you know with certainty is that the quantity will be in $[230, 240]$ mL.) Jugs B and C can be used to measure a volume in $[290, 310]$ mL and in $[500, 515]$ mL, respectively. Your recipe for chocolate chip cookies calls for at least 2100 mL and no more than 2300 mL of milk.

Write a program that determines if there exists a sequence of steps by which the required amount of milk can be obtained using the worn-out jugs. The milk is being added to a large mixing bowl, and

hence cannot be removed from the bowl. Furthermore, it is not possible to pour one jug's contents into another. Your scheme should always work, i.e., return between 2100 and 2300 mL of milk, independent of how much is chosen in each individual step, as long as that quantity satisfies the given constraints.

Hint: Solve the n jugs case.

Solution: It is natural to solve this problem using recursion—if we use jug A for the last step, we need to correctly measure a volume of milk that is at least $2100 - 230 = 1870$ mL—the last measurement may be as little as 230 mL, and anything less than 1870 mL runs the risk of being too little. Similarly, the volume must be at most $2300 - 240 = 2060$ mL. The volume is not achievable if it is not achievable with any of the three jugs as ending points. We cache intermediate computations to reduce the number of recursive calls.

In the following code, we implement a general purpose function which finds the feasibility among n jugs.

```
Jug = collections.namedtuple('Jug', ('low', 'high'))


def check_feasible(jugs, L, H, c=set()):
    VolumeRange = collections.namedtuple('VolumeRange', ('low', 'high'))
    if L > H or VolumeRange(L, H) in c or (L < 0 and H < 0):
        return False

    # Checks the volume for each jug to see if it is possible.
    if any((L <= j.low and j.high <= H)
           or check_feasible(jugs, L - j.low, H - j.high) for j in jugs):
        return True
    c.add(VolumeRange(L, H))  # Marks this as impossible.
    return False
```

The time complexity is $O((L + 1)(H + 1)n)$. The time directly spent within each call to `CheckFeasibleHelper`, except for the recursive calls, is $O(n)$, and because of the cache, there are at most $(L + 1)(H + 1)$ calls to `CheckFeasibleHelper`. The space complexity is $O((L + 1)(H + 1))$, which is the upper bound on the size of the cache.

Note that it is possible to formulate this problem using Integer Linear Programming (ILP). However, typically interviewers will not be satisfied with a reduction to ILP since such a solution does not demonstrate any programming skills.

Variant: Suppose Jug i can be used to measure any quantity in $[l_i, u_i]$ exactly. Determine if it is possible to measure a quantity of milk between L and U.

24.28 Compute the maximum subarray sum in a circular array

Finding the maximum subarray sum in an array can be solved in linear time, as described on Page 235. However, if the given array A is circular, which means the first and last elements of the array are to be treated as being adjacent to each other, the algorithm yields suboptimum solutions. For example, if A is the array in Figure 16.2 on Page 235, the maximum subarray sum starts at index 7 and ends at index 3, but the algorithm described on Page 236 returns the subarray from index 0 to index 3.

Given a circular array A, compute its maximum subarray sum in $O(n)$ time, where n is the length of A. Can you devise an algorithm that takes $O(n)$ time and $O(1)$ space?

Hint: The maximum subarray may or may not wrap around.

Solution: First recall the standard algorithm for the conventional maximum subarray sum problem. This proceeds by computing the maximum subarray sum $S[i]$ when the subarray ends at i, which is $\max(S[i-1]+A[i], A[i])$. Its time complexity is $O(n)$, where n is the length of the array, and space complexity is $O(1)$.

One approach for the maximum circular subarray is to break the problem into two separate instances. The first instance is the noncircular one, and is solved as described above.

The second instance entails looking for the maximum subarray that cycles around. Naively, this entails finding the maximum subarray that starts at index 0, the maximum subarray ending at index $n-1$, and adding their sums. However, these two subarrays may overlap, and simply subtracting out the overlap does not always give the right result (consider the array $\langle 10, -4, 5, -4, 10\rangle$).

Instead, we compute for each i the maximum subarray sum S_i for the subarray that starts at 0 and ends at or before i, and the maximum subarray E_i for the subarray that starts after i and ends at the last element. Then the maximum subarray sum for a subarray that cycles around is the maximum over all i of $S_i + E_i$.

```
def max_subarray_sum_in_circular(A):
    # Calculates the non-circular solution.
    def find_max_subarray():
        maximum_till = maximum = 0
        for a in A:
            maximum_till = max(a, a + maximum_till)
            maximum = max(maximum, maximum_till)
        return maximum

    # Calculates the solution which is circular.
    def find_circular_max_subarray():
        def compute_running_maximum(A):
            partial_sum = A[0]
            running_maximum = [partial_sum]
            for a in A[1:]:
                partial_sum += a
                running_maximum.append(max(running_maximum[-1], partial_sum))
            return running_maximum

        # Maximum subarray sum starts at index 0 and ends at or before index i.
        maximum_begin = compute_running_maximum(A)
        # Maximum subarray sum starts at index i + 1 and ends at the last
        # element.
        maximum_end = compute_running_maximum(A[::-1])[::-1][1:] + [0]

        # Calculates the maximum subarray which is circular.
        return max(begin + end
                   for begin, end in zip(maximum_begin, maximum_end))

    return max(find_max_subarray(), find_circular_max_subarray())
```

The time complexity and space complexity are both $O(n)$.

Alternatively, the maximum subarray that cycles around can be determined by computing the minimum subarray—the remaining elements yield a subarray that cycles around. (One or both of the first and last elements may not be included in this subarray, but that is fine.) This approach uses $O(1)$ space and $O(n)$ time; code for it is given below.

```
def max_subarray_sum_in_circular(A):
    def find_optimum_subarray_using_comp(comp):
        till = overall = 0
        for a in A:
            till = comp(a, a + till)
            overall = comp(overall, till)
        return overall

    # Finds the max in non-circular case and circular case.
    return max(
        find_optimum_subarray_using_comp(max),  # Non-circular case.
        sum(A) - find_optimum_subarray_using_comp(min))  # Circular case.
```

24.29 Determine the critical height

You need to test the design of a protective case. Specifically, the case can protect the enclosed device from a fall from up to some number of floors, and you want to determine what that number of floors is. You can assume the following:

- All cases have identical physical properties. In particular, if one breaks when falling from a particular level, all of them will break when falling from that level.
- A case that survives a fall can be used again, and a broken case must be discarded.
- If a case breaks when dropped, then it would break if dropped from a higher floor, and if a case survives a fall, then it would survive a shorter fall.

It is not ruled out that the first-floor windows break cases, nor is it ruled out that cases can survive the 36th-floor windows.

You know that there exists a floor such that the case will break if it is dropped from any floor at or above that floor, will remain intact if dropped from a lower floor. The ground floor is numbered zero, and it is given that the case will not break if dropped from the ground floor.

An additional constraint is that you can perform only a fixed number of drops before the building supervisor stops you.

Note that if we have a single case and are allowed only 5 drops, then the highest we can measure to is 5 floors, testing from 1, 2, 3, 4, and 5. We cannot skip a floor, since the case may break immediately after the skipped floor, and we would have no way to know if the critical floor was the last one tested or a skipped floor. If the case does not break, we know it is able to last to a fifth floor drop.

If we have two cases and are allowed 5 drops, we can do better. For example, we could test by dropping from floors 2, 4, 6, 8, 9. If a case breaks on the first four drops, we have narrowed the critical floor to that floor or the one below it. We can test the one below it with the second case. If the case breaks on the fifth drop, we know the critical floor is 9. If the case does not break, we know it is able to last to a ninth floor drop. Clearly having two cases is better than one.

Given c cases and a maximum of d allowable drops, what is the maximum number of floors that you can test in the worst-case?

Hint: Write a recurrence relation.

Solution: Let $F(c,d)$ be the maximum number of floors we can test with c identical cases and at most d drops. We know that $F(1,d) = d$. Suppose we know the value of $F(i,j)$ for all $i \leq c$ and $j \leq d$.

If we are given $c+1$ cases and d drops we can start at floor $F(c,d-1)+1$ and drop a case. If the case breaks, then we can use the remaining c cases and $d-1$ drops to determine the floor exactly, since it must be in the range $[1, F(c,d-1)]$. If the case did not break, we proceed to floor $F(c,d-1)+1+F(c+1,d-1)$.

Therefore, F satisfies the recurrence

$$F(c+1,d) = F(c,d-1)+1+F(c+1,d-1).$$

We can compute F using DP as below:

```
def get_height(cases, drops):
    def get_height_helper(cases, drops):
        if cases == 0 or drops == 0:
            return 0
        elif cases == 1:
            return drops
        if F[cases][drops] == -1:
            F[cases][drops] = (get_height_helper(cases, drops - 1) +
                               get_height_helper(cases - 1, drops - 1) + 1)
        return F[cases][drops]

    F = [[-1] * (drops + 1) for i in range(cases + 1)]
    return get_height_helper(cases, drops)
```

The time and space complexity are $O((c+1)(d+1))$.

Variant: Solve the same problem with $O(c)$ space.

Variant: How would you compute the minimum number of drops needed to find the breaking point from 1 to F floors using c cases?

Variant: Men numbered from 1 to n are arranged in a circle in clockwise order. Every kth man is removed, until only one man remains. What is the number of the last man?

24.30 Find the maximum 2D subarray

The following problem has applications to image processing.

Let A be an $n \times m$ Boolean 2D array. Design efficient algorithms for the following two problems:

- What is the largest 2D subarray containing only 1s?
- What is the largest square 2D subarray containing only 1s?

What are the time and space complexities of your algorithms as a function of n and m?

Hint: How would you efficiently check if $A[i,i+a][j,j+b]$ satisfies the constraints, assuming you have already performed similar checks?

Solution: A brute-force approach is to examine all 2D subarrays. Since a 2D subarray is characterized by two diagonally opposite corners the total number of such arrays is $O(m^2n^2)$. Each 2D subarray can be checked by examining the corresponding entries, so the overall complexity is $O(m^3n^3)$. This can be easily reduced to $O(m^2n^2)$ by processing 2D subarrays by size, and reusing results—the 2D subarray $A[i, i+a][j, j+b]$ is feasible if and only if the 2D subarrays $A[i, i+a-1][j, j+b-1]$, $A[i+a, i+a][j, j+b-1]$, $A[i, i+a-1][j+b, j+b]$, and $A[i+a, i+a][j+b, j+b]$ are feasible. This is an $O(1)$ time operation, assuming that feasibility of the smaller 2D subarrays has already been computed and stored. (Note that this solution requires $O(m^2n^2)$ storage.)

The following approach lowers the time and space complexity. For each feasible entry $A[i][j]$ we record $(h_{i,j}, w_{i,j})$, where $h_{i,j}$ is the largest L such that all the entries in $A[i, i+L-1][j, j]$ are feasible, and $w_{i,j}$ is the largest L such that all the entries in $A[i, i][j, j+L-1]$ are feasible. This computation can be performed in $O(mn)$ time, and requires $O(mn)$ storage.

Now for each feasible entry $A[i][j]$ we calculate the largest 2D subarray that has $A[i][j]$ as its bottom-left corner. We do this by processing each entry in $A[i, i+h_{i,j}-1][j, j]$. As we iterate through the entries in vertical order, we update w to the smallest $w_{i,j}$ amongst the entries processed so far. The largest 2D subarray that has $A[i][j]$ as its bottom-left corner and $A[i'][j]$ as its top-left corner has area $(i'-i+1)w$. We track the largest 2D subarray seen so far across all $A[i][j]$ processed.

```
def max_rectangle_submatrix(A):
    MaxHW = collections.namedtuple('MaxHW', ('h', 'w'))
    # DP table stores (h, w) for each (i, j).
    table = [[None] * len(A[0]) for _ in A]

    for i, row in reversed(list(enumerate(A))):
        for j, v in reversed(list(enumerate(row))):
            # Find the largest h such that (i, j) to (i + h - 1, j) are feasible.
            # Find the largest w such that (i, j) to (i, j + w - 1) are feasible.
            table[i][j] = (MaxHW(table[i + 1][j].h + 1
                                 if i + 1 < len(A) else 1, table[i][j + 1].w + 1
                                 if j + 1 < len(row) else 1)
                           if v else MaxHW(0, 0))

    max_rect_area = 0
    for i, row in enumerate(A):
        for j, v in enumerate(row):
            # Process (i, j) if it is feasible and is possible to update
            # max_rect_area.
            if v and table[i][j].w * table[i][j].h > max_rect_area:
                min_width = float('inf')
                for a in range(table[i][j].h):
                    min_width = min(min_width, table[i + a][j].w)
                    max_rect_area = max(max_rect_area, min_width * (a + 1))
    return max_rect_area
```

The time complexity per $A[i][j]$ is proportional to the number of rows, i.e., $O(n)$, yielding an overall time complexity of $O(mn^2)$, and space complexity of $O(mn)$.

If we are looking for the largest feasible square region, we can improve the complexity as follows—we compute the $(h_{i,j}, w_{i,j})$ values as before. Suppose we know the length s of the largest square region that has $A[i+1][j+1]$ as its bottom-left corner. Then the length of the side of the largest square with $A[i][j]$ as its bottom-left corner is at most $s+1$, which occurs if and only if

$h_{i,j} \geq s+1$ and $w_{i,j} \geq s+1$. The general expression for the length is $\min(s+1, h_{i,j}, w_{i,j})$. Note that this is an $O(1)$ time computation. In total, the run time is $O(mn)$, a factor of n better than before.

The calculations above can be sped up by intelligent pruning. For example, if we already have a feasible 2D subarray of dimensions $H \times W$, there is no reason to process an entry $A[i][j]$ for which $h_{i,j} \leq H$ and $w_{i,j} \leq W$.

```
def max_square_submatrix(A):
    MaxHW = collections.namedtuple('MaxHW', ('h', 'w'))
    # DP table stores (h, w) for each (i, j).
    table = [[None] * len(A[0]) for _ in A]
    for i, row in reversed(list(enumerate(A))):
        for j, v in reversed(list(enumerate(row))):
            # Finds the largest h such that (i, j) to (i + h - 1, j) are feasible.
            # Finds the largest w such that (i, j) to (i, j + w - 1) are feasible.
            table[i][j] = (MaxHW(table[i + 1][j].h + 1
                                 if i + 1 < len(A) else 1, table[i][j + 1].w + 1
                                 if j + 1 < len(row) else 1)
                           if v else MaxHW(0, 0))

    # A table stores the length of the largest square for each (i, j).
    s = [[0] * len(A[0]) for _ in A]
    max_square_area = 0
    for i, row in reversed(list(enumerate(A))):
        for j, v in reversed(list(enumerate(row))):
            if v:
                side = min(table[i][j].h, table[i][j].w)
                # Gets the length of largest square with bottom-left corner (i,
                # j).
                if i + 1 < len(A) and j + 1 < len(A[i + 1]):
                    side = min(s[i + 1][j + 1] + 1, side)
                s[i][j] = side
                max_square_area = max(max_square_area, side**2)
    return max_square_area
```

The largest 2D subarray can be found in $O(nm)$ time using a qualitatively different approach. Essentially, we reduce our problem to n instances of the largest rectangle under the skyline problem described in Problem 17.8 on Page 270. First, for each $A[i][j]$ we determine the largest $h_{i,j}$ such that $A[i, i + h_{i,j} - 1][j, j]$ is feasible. (If $A[i][j] = 0$ then $h_{i,j} = 0$.) Then for each of the n rows, starting with the topmost one, we compute the largest 2D subarray whose bottom edge is on that row in time $O(m)$, using Solution 17.8 on Page 271. This computation can be performed in time $O(n)$ once the $h_{i,j}$ values have been computed. The final solution is the maximum of the n instances.

The time complexity for each row is $O(m)$ for computing the h values, assuming we record the h values for the previous row, and $O(m)$ for computing the largest rectangle under the skyline, i.e., $O(m)$ in total per row. Therefore the total time complexity is $O(mn)$. The additional space complexity is $O(m)$—this is the space for recording the h values and running the largest rectangle under the skyline computation.

```
def max_rectangle_submatrix(A):
    table = [0] * len(A[0])
    max_rect_area = 0
    # Find the maximum among all instances of the largest rectangle.
    for row in A:
```

```
        table = [x + y if y else 0 for x, y in zip(table, row)]
        max_rect_area = max(max_rect_area, calculate_largest_rectangle(table))
    return max_rect_area
```

The largest square 2D subarray containing only 1s can be computed similarly, with a minor variant on the algorithm in Solution 17.8 on Page 271.

Variant: Solve the largest square 2D subarray problem using $O(m)$ time.

24.31 Implement Huffman coding

One way to compress text is by building a code book which maps each character to a bit string, referred to as its code word. Compression consists of concatenating the bit strings for each character to form a bit string for the entire text. (The codebook, i.e., the mapping from characters to corresponding bit strings is stored separately, e.g., in a preamble.)

When decompressing the string, we read bits until we find a string that is in the code book and then repeat this process until the entire text is decoded. For the compression to be reversible, it is sufficient that the code words have the property that no code word is a prefix of another. For example, 011 is a prefix of 0110 but not a prefix of 1100.

Since our objective is to compress the text, we would like to assign the shorter bit strings to more common characters and the longer bit strings to less common characters. We will restrict our attention to individual characters. (We may achieve better compression if we examine common sequences of characters, but this increases the time complexity.)

The intuitive notion of commonness is formalized by the *frequency* of a character which is a number between zero and one. The sum of the frequencies of all the characters is 1. The average code length is defined to be the sum of the product of the length of each character's code word with that character's frequency. Table 24.1 on the following page shows the frequencies of letters of the English alphabet.

Table 24.1: English characters and their frequencies, expressed as percentages, in everyday documents.

Character	Frequency	Character	Frequency	Character	Frequency
a	8.17	j	0.15	s	6.33
b	1.49	k	0.77	t	9.06
c	2.78	l	4.03	u	2.76
d	4.25	m	2.41	v	0.98
e	12.70	n	6.75	w	2.36
f	2.23	o	7.51	x	0.15
g	2.02	p	1.93	y	1.97
h	6.09	q	0.10	z	0.07
i	6.97	r	5.99		

Given a set of characters with corresponding frequencies, find a code book that has the smallest average code length.

Hint: Reduce the problem from n characters to one on $n - 1$ characters.

Solution: The trivial solution is to use fixed length bit strings for each character. To be precise, if there are n distinct characters, we can use $\lceil \log n \rceil$ bits per character. If all characters are equally likely, this is optimum, but when there is large variation in frequencies, we can do much better.

A natural heuristic is to split the set of characters into two subsets, which have approximately equal aggregate frequencies, solve the problem for each subset, and then add a 0 to the codes from the first set and a 1 to the codes from the second set to differentiate the codes from the two subsets. This approach does not always result in the optimum coding, e.g., when the characters are A, B, C, D with frequencies $\{0.4, 0.35, 0.2, 0.05\}$, it forms the code words $00, 10, 11, 01$, whereas the optimum coding is $0, 10, 110, 111$. It also requires being able to partition the characters into two subsets whose aggregate frequencies are close, which is computationally challenging.

Another strategy is to assign 0 to the character with highest frequency, solve the same problem on the remaining characters, and then prefix those codes with a 1. This approach fares very poorly when characters have the same frequency, e.g., if A, B, C, D all have frequency 0.25, the resulting coding is $0, 10, 100, 111$, whereas $00, 01, 10, 11$ is the optimum coding. Intuitively, this strategy fails because it does not take into account the relative frequencies.

Huffman coding yields an optimum solution to this problem. (There may be other optimum codes as well.) It is based on the idea that you should focus on the least frequent characters, rather than the most frequent ones. Specifically, combine the two least frequent characters into a new character, recursively solve the problem on the resulting $n - 1$ characters; then create codes for the two combined characters from the code for their combined character by adding a 0 for one and a 1 for the other.

More precisely, Huffman coding proceeds in three steps:

(1.) Sort characters in increasing order of frequencies and create a binary tree node for each character. Denote the set just created by S.

(2.) Create a new node u whose children are the two nodes with smallest frequencies and assign u's frequency to be the sum of the frequencies of its children.

(3.) Remove the children from S and add u to S. Repeat from Step (2.) till S consists of a single node, which is the root.

Mark all the left edges with 0 and the right edges with 1. The path from the root to a leaf node yields the bit string encoding the corresponding character.

Applying this algorithm to the frequencies for English characters presented in Table 24.1 yields the Huffman tree in Figure 24.9 on the next page. The path from root to leaf yields that character's Huffman code, which is listed in Table 24.2 on the facing page. For example, the codes for t, e, and z are 000, 100, and 001001000, respectively.

The codebook is explicitly given in Table 24.2 on the next page. The average code length for this coding is 4.205. In contrast, the trivial coding takes $\lceil \log 26 \rceil = 5$ bits for each character.

In the implementation below, we use a min-heap of candidate nodes to represent S.

```
CharWithFrequency = collections.namedtuple('CharWithFrequency', ('c', 'freq'))


def huffman_encoding(symbols):
    class BinaryTree:
        def __init__(self, aggregate_freq, s, left=None, right=None):
            self.aggregate_freq = aggregate_freq
            self.s = s
```

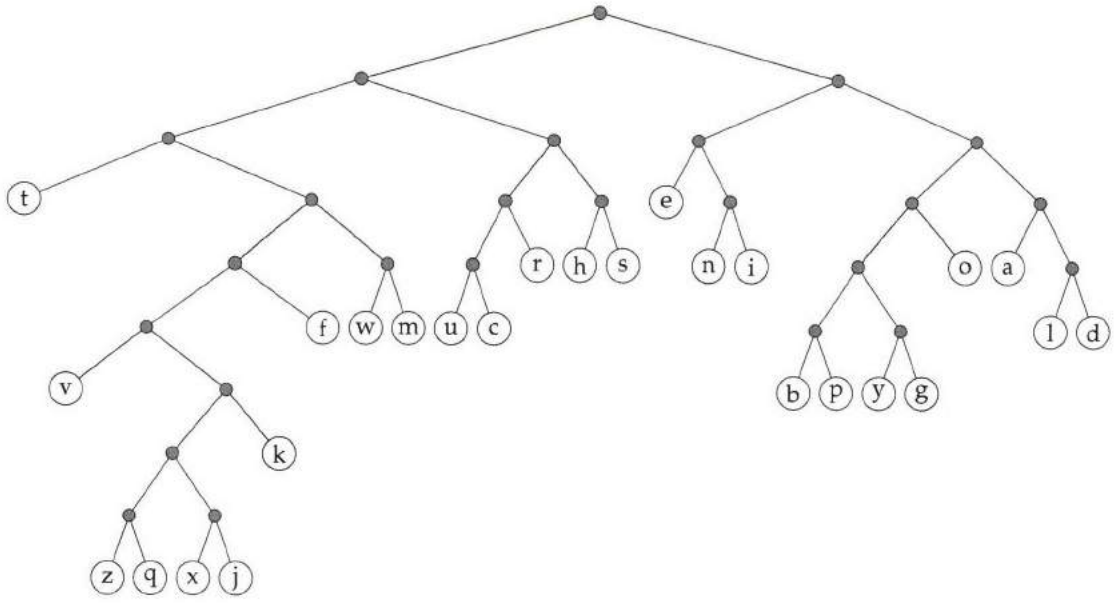


Figure 24.9: A Huffman tree for the English characters, assuming the frequencies given in Table 24.1 on the facing page.

Table 24.2: Huffman codes for English characters, assuming the frequencies given in Table 24.1 on the facing page.

Character	Huffman code	Character	Huffman code	Character	Huffman code
a	1110	j	001001011	s	0111
b	110000	k	0010011	t	000
c	01001	l	11110	u	01000
d	11111	m	00111	v	001000
e	100	n	1010	w	00110
f	00101	o	1101	x	001001010
g	110011	p	110001	y	110010
h	0110	q	001001001	z	001001000
i	1011	r	0101		

```
        self.left, self.right = left, right

    def __lt__(self, other):
        return self.aggregate_freq <= other.aggregate_freq
# Initially assigns each symbol into candidates.
candidates = [BinaryTree(s.freq, s) for s in symbols]
heapq.heapify(candidates)

# Keeps combining two nodes until there is one node left.
while len(candidates) > 1:
    left, right = heapq.heappop(candidates), heapq.heappop(candidates)
    heapq.heappush(candidates,
                   BinaryTree(left.aggregate_freq + right.aggregate_freq,
                              None, left, right))

def assign_huffman_code(tree, code):
    if tree:
        if tree.s:
```

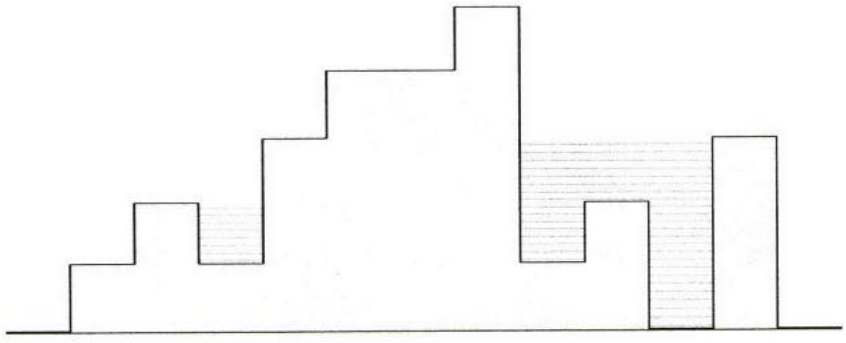


Figure 24.10: The area of the horizontal lines is the maximum amount of water that can be trapped by the solid region. For this container, it is $1 + 2 + 1 + 3 = 7$.

```
            # This node is a leaf.
            huffman_encoding[tree.s.c] = ''.join(code)
        else:  # Non-leaf node.
            code.append('0')
            assign_huffman_code(tree.left, code)
            code[-1] = '1'
            assign_huffman_code(tree.right, code)
            del code[-1]

    huffman_encoding = {}
    # Traverses the binary tree, assigning codes to nodes.
    assign_huffman_code(candidates[0], [])
    return sum(len(huffman_encoding[s.c]) * s.freq / 100 for s in symbols)
```

Since each invocation of Steps (2.) on Page 396 and (3.) on Page 396 requires two *extract-min* and one *insert* operation, it takes $O(n \log n)$ time to build the Huffman tree, where n is the number of characters. It's possible for the tree to be very skewed. In such a situation, the codewords are of length $1, 2, 3, \ldots, n$, so the time to generate the codebook becomes $O(1 + 2 + \cdots + n) = O(n^2)$.

It is exceedingly unlikely that you would be asked for a rigorous proof of optimality in an interview setting. The proof that the Huffman algorithm yields the minimum average code length uses induction on the number of characters. The induction step itself makes use of proof by contradiction, with the two leaves in the Huffman tree corresponding to the rarest characters playing a central role.

24.32 Trapping water

The goal of this problem is to compute the capacity of a type of one-dimensional container. The computation is illustrated in Figure 24.10.

A one-dimensional container is specified by an array of n nonnegative integers, specifying the height of each unit-width rectangle. Design an algorithm for computing the capacity of the container.

Hint: Draw pictures, and focus on the extremes.

Solution: We can get a great deal of insight by visualizing pouring water into the container. When the maximum capacity is achieved, the cross-section consists of a region in which the water level is

nondecreasing, followed by a region in which the water level is nonincreasing. The transition from nondecreasing to nonincreasing must take place around the maximum entry in A. Let $A[m]$ be a maximum value entry. Then we compute the capacity of $A[0, m-1]$ and $A[m, n-1]$ independently. These capacities are determined via an iteration. For each entry in $A[0, m-1]$ we compute the difference between its value entry and the running maximum, and add that to the total capacity. We handle $A[m, n-1]$ analogously. The time complexity is $O(n)$ to find a maximum, and then $O(n)$ for each of the two iterations, i.e., the total time complexity is $O(n)$. The space complexity is $O(1)$—all that is needed is to record several variables.

```
def calculate_trapping_water(heights):
    # Finds the index with maximum height.
    max_h = heights.index(max(heights))

    # Assume heights[-1] is maximum height.
    def trapping_water_till_end(heights):
        partial_sum, highest_level_seen = 0, float('-inf')
        for h in heights:
            if h >= highest_level_seen:
                highest_level_seen = h
            else:
                partial_sum += highest_level_seen - h
        return partial_sum

    return (trapping_water_till_end(heights[:max_h]) +
            trapping_water_till_end(reversed(heights[max_h + 1:])))
```

Variant: Solve the water filling problem with an algorithm that accesses A's elements in order and can read an element only once. Use minimum additional space.

24.33 The heavy hitter problem

This problem is a generalization of Problem 17.5 on Page 266. In practice we may not be interested in a majority token but all tokens whose count exceeds say 1% of the total token count. It is fairly straightforward to show that it is impossible to compute such tokens in a single pass when you have limited memory. However, if you are allowed to pass through the sequence twice, it is possible to identify the common tokens.

You are reading a sequence of strings separated by whitespace. You are allowed to read the sequence twice. Devise an algorithm that uses $O(k)$ memory to identify the words that occur more than $\frac{n}{k}$ times, where n is the length of the sequence.

Hint: Maintain a list of k candidates.

Solution: This is essentially a generalization of Problem 17.5 on Page 266. Here instead of discarding two distinct words, we discard k distinct words at any given time and we are guaranteed that all the words that occurred more than $\frac{1}{k}$ times the length of the sequence prior to discarding continue to appear more than $\frac{1}{k}$ times in the remaining sequence. To implement this strategy, we need a hash table of the current candidates.

```
# Finds the candidates which may occur > n / k times.
```

```
def search_frequent_items(k, stream):
    table = collections.Counter()
    n = 0  # Count the number of strings.

    for buf in stream:
        table[buf] += 1
        n += 1
        # Detecting k items in table, at least one of them must have exactly
        # one in it. We will discard those k items by one for each.
        if len(table) == k:
            for it in table:
                table[it] -= 1
            table = +table  # remove all zero values

    # Resets table for the following counting.
    for it in table:
        table[it] = 0

    # Resets the stream and read it again.
    # Counts the occurrence of each candidate word.
    for buf in stream:
        if buf in table:
            table[buf] += 1

    # Selects the word which occurs > n / k times.
    return [it for it, value in table.items() if value > n / k]
```

The code may appear to take $O(nk)$ time since the inner loop may take k steps (decrementing count for all k entries) and the outer loop is called n times. However each word in the sequence can be erased only once, so the total time spent erasing is $O(n)$ and the rest of the steps inside the outer loop run in $O(1)$ time.

The first step yields a set S of not more than k words; set S is a superset of the words that occur greater than $\frac{n}{k}$ times. To get the exact set, we need to make another pass over the sequence and count the number of times each word in S actually occurs. We return the words in S which occur more than $\frac{n}{k}$ times.

24.34 Find the longest subarray whose sum $\leq k$

Here we consider finding the longest subarray subject to a constraint on the subarray sum. For example, for the array in Figure 24.11 on the next page, the longest subarray whose subarray sum is no more than 184 is $A[3, 6]$.

431	-15	639	342	-14	565	-924	635	167	-70
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 24.11: An array for the longest subarray whose sum $\leq k$ problem.

Design an algorithm that takes as input an array A of n numbers and a key k, and returns the length of a longest subarray of A for which the subarray sum is less than or equal to k.

Hint: When can you be sure that an index i cannot be the starting point of a subarray with the desired property, without looking past i?

Solution: The brute-force solution entails computing $\sum_{k=i}^{j} A[k]$, i.e., the sum of the element in $A[i, j]$, for all $0 \leq i \leq j \leq n-1$, where n is the length of A. Let P be the prefix sum array for A, i.e., $P[i] = \sum_{k=0}^{i} A[k]$; P can be computed in a single iteration over A in $O(n)$ time. Note that the sum of the elements in the subarray $A[i, j]$ is $P[j] - P[i-1]$ (for convenience, take $P[-1] = 0$). The time complexity of the brute-force solution is $O(n^2)$, and the additional space complexity is $O(n)$ (the size of P).

The trick to improving the time complexity to $O(n)$ comes from the following observation. Suppose $u < v$ and $P[u] \geq P[v]$. Then u will never be the ending point of a solution. The reason is that for any $w \leq u$, $A[w, v]$ is longer than $A[w, u]$ and if $A[w, u]$ satisfies the sum constraint, so must $A[w, v]$. This motivates the definition of the array Q: set $Q[i] = \min(P[i], Q[i+1])$ for $i < n-1$, and $Q[n-1] = P[n-1]$.

Let $a \leq b$ be indices of elements in A. Define $M_{a,b}$ to be the minimum possible sum of a subarray beginning at a and extending to b or beyond. Note that $M_{0,b} = Q[b]$, and $M_{a,b} = Q[b] - P[a-1]$, when $a > 0$. If $M_{a,b} > k$, no subarray starting at a that includes b can satisfy the sum constraint, so we can increment a. If $M_{a,b} \leq k$, then we are assured there exists a subarray of length $b - a + 1$ satisfying the sum constraint, so we compare the length of the longest subarray satisfying the sum constraint identified so far to $b - a + 1$ and conditionally update it. Consequently, we can increment b.

Suppose we initialize a and b to 0 and iteratively perform the increments to a and b described above until $b = n$. Then we will discover the length of a maximum length subarray that satisfies the sum constraint. We justify this claim after presenting an implementation of these ideas below.

```
def find_longest_subarray_less_equal_k(A, k):
    # Builds the prefix sum according to A.
    prefix_sum = list(itertools.accumulate(A))

    # Early returns if the sum of A is smaller than or equal to k.
    if prefix_sum[-1] <= k:
        return len(A)

    # Builds min_prefix_sum.
    min_prefix_sum = list(
        reversed(
            functools.reduce(lambda s, v: s + [min(v, s[-1])],
                             reversed(prefix_sum[:-1]), [prefix_sum[-1]])))
    a = b = max_length = 0
    while a < len(A) and b < len(A):
        min_curr_sum = (min_prefix_sum[b] - prefix_sum[a - 1]
                        if a > 0 else min_prefix_sum[b])
        if min_curr_sum <= k:
            curr_length = b - a + 1
            if curr_length > max_length:
                max_length = curr_length
            b += 1
        else:  # min_curr_sum > k.
```

```
        a += 1
return max_length
```

Now we argue the correctness of the program. Let $A[a^*, b^*]$ be a maximum length subarray that satisfies the sum constraint. Note that we increment b until $M_{a,b} > k$. In particular, when we increment a to $a+1$, $A[a, b-1]$ does satisfy the sum constraint, but $A[a, b]$ does not. This implies $A[a, b-1]$ is the longest subarray starting at a that satisfies the sum constraint.

The iteration ends when $b = n$. At this point, we claim $a \geq a^*$. If not, then $A[a, n-1]$ satisfies the sum constraint, since we incremented b to n, and $(n-1) - a + 1 > b^* - a^* + 1$, contradicting the optimality of $A[a^*, b^*]$. Therefore, a must be assigned to a^* at some iteration. At this point, $b \leq b^*$ since $A[a^*-1, b-1]$ satisfies the sum constraint. For, if $b > b^*$, then $(b-1)-(a^*-1)+1 = b-a^*+1 > b^*-a^*+1$, violating the maximality of $A[a^*, b^*]$. Since $b \leq b^*$ and $a = a^*$, the algorithm will increment b till it becomes b^* (since $A[a^*, b^*]$ satisfies the sum constraint), and thus will identify $b^* - a^* + 1$ as the optimum solution.

Variant: Design an algorithm for finding the longest subarray of a given array such that the average of the subarray elements is $\leq k$.

24.35 Road network

The California Department of Transportation is considering adding a new section of highway to the California Highway System. Each highway section connects two cities. City officials have submitted proposals for the new highway—each proposal includes the pair of cities being connected and the length of the section.

Write a program which takes the existing highway network (specified as a set of highway sections between pairs of cities) and proposals for new highway sections, and returns the proposed highway section which leads to the most improvement in the total driving distance. The total driving distance is defined to be the sum of the shortest path distances between all pairs of cities. All sections, existing and proposed, allow for bi-directional traffic, and the original network is connected.

Hint: Suppose we add a new section from b_s to b_f. If the shortest path from u to v passes through this section, what must be true of the part of the path from u to b_s?

Solution: Note that we cannot add more than one proposal to the existing network and run a shortest path algorithm—we may end up with a shortest path which uses multiple proposals.

The brute-force approach would be to first compute the shortest path distances for all pairs in the original network. Then consider the new sections, one-at-a-time, and then compute the new shortest path distances for all pairs, recording the total improvement. The all-pairs shortest path problem can be solved in time $O(n^3)$ using the Floyd-Warshall algorithm, leading to an overall $O(kn^3)$ time complexity.

We can improve upon this by running the all pairs shortest paths algorithm just once. Let $S(u, v)$ be the 2D array of shortest path distances for each pair of cities. Each proposal p is a pair of cities (x, y). For the pair of cities (a, b), the best we can do by using proposal p is $\min(S(a, b), S(a, x)+d(x, y)+S(y, b), S(a, y)+d(y, x)+S(x, b))$ where $d(x, y)$ is the distance of the proposed highway p between x and y. This computation is $O(1)$ time, so we can evaluate all the proposals in time proportional to the

number of proposals times the number of pairs after we have computed the shortest path between each pair of cities. This results in an $O(n^3 + kn^2)$ time complexity, which improves substantially on the brute-force approach.

```
HighwaySection = collections.namedtuple('HighwaySection', ('x', 'y',
                                                           'distance'))


def find_best_proposals(H, P, n):
    # G stores the shortest path distances between all pairs of vertices.
    G = [[float('inf')] * i + [0] + [float('inf')] * (n - i - 1)
         for i in range(n)]
    # Builds an undirected graph G based on existing highway sections H.
    for h in H:
        G[h.x][h.y] = G[h.y][h.x] = h.distance

    def floyd_warshall(G):
        for k, i, j in itertools.product(range(len(G)), repeat=3):
            if G[i][k] != float('inf') and G[k][j] != float('inf'):
                G[i][j] = min(G[i][j], G[i][k] + G[k][j])

    # Performs floyd warshall to build the shortest path between vertices.
    floyd_warshall(G)

    # Examines each proposal for shorter distance for all pairs.
    best_distance_saving = float('-inf')
    best_proposal = HighwaySection(-1, -1, 0)  # Default.
    for p in P:
        proposal_saving = 0
        for a, b in itertools.product(range(n), repeat=2):
            saving = G[a][b] - (G[a][p.x] + p.distance + G[p.y][b])
            proposal_saving += max(saving, 0)
        if proposal_saving > best_distance_saving:
            best_distance_saving = proposal_saving
            best_proposal = p
    return best_proposal
```

24.36 Test if arbitrage is possible

Suppose you are given a set of exchange rates among currencies and you want to determine if an arbitrage exists, i.e., there is a way by which you can start with one unit of some currency C and perform a series of barters which results in having more than one unit of C. Transaction costs are zero, exchange rates do not fluctuate, fractional quantities of items can be sold, and the exchange rate between each pair of currencies is finite.

Table 24.3 shows a representative example. An arbitrage is possible for this set of exchange rates: 1 USD $\rightarrow 1 \times 0.8123 = 0.8123$ EUR $\rightarrow 0.8123 \times 1.2010 = 0.9755723$ CHF $\rightarrow 0.9755723 \times 80.39 = 78.426257197$ JPY $\rightarrow 78.426257197 \times 0.0128 = 1.00385609212$ USD.

Design an efficient algorithm to determine whether there exists an arbitrage—a way to start with a single unit of some currency C and convert it back to more than one unit of C through a sequence of exchanges.

Table 24.3: Exchange rates for seven major currencies.

Symbol	USD	EUR	GBP	JPY	CHF	CAD	AUD
USD	1	0.8123	0.6404	78.125	0.9784	0.9924	0.9465
EUR	1.2275	1	0.7860	96.55	1.2010	1.2182	1.1616
GBP	1.5617	1.2724	1	122.83	1.5280	1.5498	1.4778
JPY	0.0128	0.0104	0.0081	1	1.2442	0.0126	0.0120
CHF	1.0219	0.8327	0.6546	80.39	1	1.0142	0.9672
CAD	1.0076	0.8206	0.6453	79.26	0.9859	1	0.9535
AUD	1.0567	0.8609	0.6767	83.12	1.0339	1.0487	1

Hint: The effect of a sequence of conversions is multiplicative. Can you recast the problem so that it can be calculated additively?

Solution: Drawing pictures is a great way to brainstorm for a potential solution. In this case, the relationships between the currencies can be represented using a graph. Specifically, we can model the problem with a graph where currencies correspond to vertices, exchanges correspond to edges, and the edge weight is set to the logarithm of the exchange rate. If we can find a cycle in the graph with a positive weight, we would have found such a series of exchanges. Such a cycle can be solved using the Bellman-Ford algorithm.

We define a weighted directed graph $G = (V, E = V \times V)$, where V corresponds to the set of currencies. The weight $w(e)$ of edge $e = (u, v)$ is the amount of currency v we can buy with one unit of currency u. Observe that an arbitrage exists if and only if there exists a cycle in G whose edge weights multiply out to more than 1.

Create a new graph $G' = (V, E)$ with weight function $w'(e) = -\log w(e)$. Since $\log(a \times b) = \log a + \log b$, there exists a cycle in G whose edge weights multiply out to more than 1 if and only if there exists a cycle in G' whose edge weights sum up to less than $\log 1 = 0$. (This property is true for logarithms to any base, so if it is more efficient for example to use base-e, we can do so.)

The Bellman-Ford algorithm detects negative-weight cycles. Usually, finding a negative-weight cycle is done by adding a dummy vertex s with 0-weight edges to each vertex in the given graph and running the Bellman-Ford single-source shortest path algorithm from s. However, for the arbitrage problem, the graph is complete. Hence, we can run Bellman-Ford algorithm from any single vertex, and get the right result.

```
def is_arbitrage_exist(G):
    def bellman_ford(G, source):
        dis_to_source = ([float('inf')] * (source - 1) + [0] + [float('inf')] *
                         (len(G) - source))

        for _ in range(1, len(G)):
            have_update = False
            for i, row in enumerate(G):
                for j, g in enumerate(row):
                    if (dis_to_source[i] != float('inf')
                            and dis_to_source[j] > dis_to_source[i] + g):
                        have_update = True
                        dis_to_source[j] = dis_to_source[i] + g

            # No update in this iteration means no negative cycle.
            if not have_update:
```

```
            return False

    # Detects cycle if there is any further update.
    return any(dis_to_source[i] != float('inf')
               and dis_to_source[j] > dis_to_source[i] + g
               for i, row in enumerate(G) for j, g in enumerate(row))

# Uses Bellman-ford to find negative weight cycle.
return bellman_ford([[-math.log10(edge) for edge in edge_list]
                     for edge_list in G], 0)
```

The time complexity of the general Bellman-Ford algorithm is $O(|V||E|)$. Here, $|E| = O(|V|^2)$ and $|V| = n$, so the time complexity is $O(n^3)$.

Part V

Notation and Index

Notation

To speak about notation as the only way that you can guarantee structure of course is already very suspect.

— E. S. Parker

We use the following convention for symbols, unless the surrounding text specifies otherwise:

- A k-dimensional array
- L linked list or doubly linked list
- S set
- T tree
- G graph
- V set of vertices of a graph
- E set of edges of a graph

Symbolism	**Meaning**
$(d_{k-1} \dots d_0)_r$	radix-r representation of a number, e.g., $(1011)_2$
$\log_b x$	logarithm of x to the base b; if b is not specified, $b = 2$
$\lvert S \rvert$	cardinality of set S
$S \setminus T$	set difference, i.e., $S \cap T'$, sometimes written as $S - T$
$\lvert x \rvert$	absolute value of x
$\lfloor x \rfloor$	greatest integer less than or equal to x
$\lceil x \rceil$	smallest integer greater than or equal to x
$\langle a_0, a_1, \dots, a_{n-1} \rangle$	sequence of n elements
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that relation $R(k)$ is true
$\min_{R(k)} f(k)$	minimum of all $f(k)$ such that relation $R(k)$ is true
$\max_{R(k)} f(k)$	maximum of all $f(k)$ such that relation $R(k)$ is true
$\sum_{k=a}^{b} f(k)$	shorthand for $\sum_{a \le k \le b} f(k)$
$\{a \mid R(a)\}$	set of all a such that the relation $R(a) =$ true
$[l, r]$	closed interval: $\{x \mid l \le x \le r\}$
$[l, r)$	left-closed, right-open interval: $\{x \mid l \le x < r\}$
$\{a, b, \dots\}$	well-defined collection of elements, i.e., a set
A_i or $A[i]$	the ith element of one-dimensional array A
$A[i, j]$	subarray of one-dimensional array A consisting of elements at indices i to j inclusive
$A[i][j]$	the element in ith row and jth column of 2D array A

$A[i_1, i_2][j_1, j_2]$	2D subarray of 2D array A consisting of elements from i_1th to i_2th rows and from j_1th to j_2th column, inclusive
$\binom{n}{k}$	binomial coefficient: number of ways of choosing k elements from a set of n items
$n!$	n-factorial, the product of the integers from 1 to n, inclusive
$O\big(f(n)\big)$	big-oh complexity of $f(n)$, asymptotic upper bound
$x \bmod y$	mod function
$x \oplus y$	bitwise-XOR function
$x \approx y$	x is approximately equal to y
`null`	pointer value reserved for indicating that the pointer does not refer to a valid address
$\emptyset$	empty set
∞	infinity: Informally, a number larger than any number.
$x \ll y$	much less than
$x \gg y$	much greater than
$\Rightarrow$	logical implication

Index of Terms

Acknowledgments

Several of our friends, colleagues, and readers gave feedback. We would like to thank Taras Bobrovytsky, Senthil Chellappan, Yi-Ting Chen, Monica Farkash, Dongbo Hu, Jing-Tang Keith Jang, Matthieu Jeanson, Gerson Kurz, Danyu Liu, Hari Mony, Shaun Phillips, Gayatri Ramachandran, Ulises Reyes, Kumud Sanwal, Tom Shiple, Ian Varley, Shaohua Wan, Don Wong, and Xiang Wu for their input.

I, Adnan Aziz, thank my teachers, friends, and students from IIT Kanpur, UC Berkeley, and UT Austin for having nurtured my passion for programming. I especially thank my friends Vineet Gupta, Tom Shiple, and Vigyan Singhal, and my teachers Robert Solovay, Robert Brayton, Richard Karp, Raimund Seidel, and Somenath Biswas, for all that they taught me. My coauthor, Tsung-Hsien Lee, brought a passion that was infectious and inspirational. My coauthor, Amit Prakash, has been a wonderful collaborator for many years—this book is a testament to his intellect, creativity, and enthusiasm. I look forward to a lifelong collaboration with both of them.

I, Tsung-Hsien Lee, would like to thank my coauthors, Adnan Aziz and Amit Prakash, who give me this once-in-a-life-time opportunity. I also thank my teachers Wen-Lian Hsu, Ren-Song Tsay, Biing-Feng Wang, and Ting-Chi Wang for having initiated and nurtured my passion for computer science in general, and algorithms in particular. I would like to thank my friends Cheng-Yi He, Da-Cheng Juan, Chien-Hsin Lin, and Chih-Chiang Yu, who accompanied me on the road of savoring the joy of programming contests; and Kuan-Chieh Chen, Chun-Cheng Chou, Ray Chuang, Wilson Hong, Wei-Lun Hung, Nigel Liang, and Huan-Kai Peng, who give me valuable feedback on this book. Last, I would like to thank all my friends and colleagues at Uber, Google, Facebook, National Tsing Hua University, and UT Austin for the brain-storming on puzzles; it is indeed my greatest honor to have known all of you.

I, Amit Prakash, have my coauthor and mentor, Adnan Aziz, to thank the most for this book. To a great extent, my problem solving skills have been shaped by Adnan. There have been occasions in life when I would not have made it through without his help. He is also the best possible collaborator I can think of for any intellectual endeavor. I have come to know Tsung-Hsien through working on this book. He has been a great coauthor. His passion and commitment to excellence can be seen everywhere in this book. Over the years, I have been fortunate to have had great teachers at IIT Kanpur and UT Austin. I would especially like to thank my teachers Scott Nettles, Vijaya Ramachandran, and Gustavo de Veciana. I would also like to thank my friends and colleagues at Google, Microsoft, IIT Kanpur, and UT Austin for many stimulating conversations and problem solving sessions. Finally, and most importantly, I want to thank my family who have been a constant source of support, excitement, and joy all my life and especially during the process of writing this book.

Adnan Aziz — *Palo Alto, California*
Tsung-Hsien Lee — *Mountain View, California*
Amit Prakash — *Saratoga, California*
October 11, 2017

CPSIA information can be obtained
at www.ICGtesting.com
Printed in the USA
LVOW04s1650041117
554956LV00005B/12/P

ELEMENTS OF PROGRAMMING INTERVIEWS IN python

"A practical, fun approach to computer science fundamentals, as seen through the lens of common programming interview questions."
—Jeff Atwood / Co-founder, Stack Overflow and Discourse

"This book prepares the reader for contemporary software interviews, and also provides a window into how algorithmic techniques translate into the workplace. It emphasizes problems that stem from real-world applications and can be coded up in a reasonable time, and is a wonderful complement to a traditional computer science algorithms and data structures course."
—Ashish Goel / Professor, Stanford University

"A wonderful resource for anyone preparing for a modern software engineering interview: work through the entire book, and you'll find the actual interview a breeze. More generally, for algorithms enthusiasts, EPI offers endless hours of entertainment while simultaneously learning neat coding tricks."
—Vineet Gupta / Principal Engineer, Google

EPI is your comprehensive guide to interviewing for software development roles.

The book begins with a summary of the nontechnical aspects of interviewing, such as strategies for a great interview, common mistakes, perspectives from the other side of the table, tips on negotiating the best offer, and a guide to the best ways to use EPI. We also provide a summary of data structures, algorithms, and problem solving patterns.

Coding problems are presented through a series of chapters on basic and advanced data structures, searching, sorting, algorithm design principles, and concurrency. Each chapter starts with a brief introduction, a case study, top tips, and a review of the most important library methods. This is followed by a broad and thought-provoking set of problems.

Adnan, Tsung-Hsien, and Amit have worked at Google, Facebook, Uber, Microsoft, IBM, Qualcomm, and several start-ups. They co-developed algorithms and systems that are used by billions of people everyday. They have extensive experience with interviewing candidates, making hiring decisions, and being interviewed.

elementsofprogramminginterviews.com

ISBN 9781537713946