CSCI 1101: Computer Science I

Jean-Baptiste Tristan

Spring 2022

Welcome to CSCI 1101: Computer Science I. Here's some important information:

- The course webpage is: `https://bostoncollege.instructure.com/courses/1627647`
- Social office hours take place in CS Lab 122 on Thursdays from 5:30-9:30pm and Fridays from 3-6pm. Feel free to drop by and ask questions or simply work on the homework!
- Everyone's contact information is below. Please remember to contact **your own** discussion section leader for technical questions (who may escalate to Julian if needed) and to contact Professor Tristan for personal questions.

Jean-Baptiste Tristan	Instructor	tristanj@bc.edu
Julian Asilis	Head TA	asilisj@bc.edu
Ananya Barthakur	TA	barthaka@bc.edu
Joseph D'Alonzo	TA	dalonzoj@bc.edu
Jakob Weiss	TA	weissjy@bc.edu
Joanne (Jo) Lee	TA	leebpo@bc.edu
Thomas Flatley	TA	flatleyt@bc.edu
Chris Conyers	TA	conyerch@bc.edu
Brielle Donowho	TA	donowhob@bc.edu

- These notes were taken by Julian and have **not been carefully proofread** – they're sure to contain some typos and omissions, due to Julian.

Contents

§1 Wednesday, January 19

Welcome to the Introduction to Computer Science. The plan today is mostly to talk about the structure of the course – rather than diving headfirst into the course material – and to talk about the spirit of computer science at a high level. In particular, we'll try to convince you that it's useful to learn about computer science even if you don't intend to become a professional programmer.

§1.1 What is computer science?

First things first: what is the definition of **computer science**? Here's what the dictionary says:

> *The branch of knowledge concerned with the construction, programming, operation, and use of computers.*

Well, what's a **computer**? Let's use the dictionary again:

> *A device or machine for performing or facilitating calculation.*

It's important to note that there's no mention of electronics here! So even something like an abacus is a computer under this definition. There's another dictionary definition of a computer though:

> *An electronic device [. . .] capable of [. . .] processing [. . .] in accordance with variable procedural instructions.*

The latter end of that definition seems to be referring to an **algorithm**, perhaps the single most important object in computer science. Let's see a definition:

> *A procedure or set of rules used in calculation and problem-solving; (in later use spec.) a precisely defined set of mathematical or logical operations for the performance of a particular task.*

The important (and difficult!) part is that algorithms are a very precise set of instructions. Defining exactly what it means to be 'precise' or 'mathematical' is no small feat, though. Formalizing this entire setup (computer, algorithm, etc.) is actually one of the most important feats of the legendary Alan Turing.

§1.2 History of computer science

One of the earliest sets of instructions for performing a computation comes from the Babylonian Empire, circa 1600 B.C. The algorithm (in more modern language) served to calculate certain dimension of a cistern. It was really a flushed out example – rather than an abstract algorithm in the modern sense – but it's thought of as one of the earliest examples of computational thinking.

In ancient Greece (circa 300 BC), Euclid developed an algorithm to compute the greatest common divisor of two numbers. This was a fairly flushed out example, and an important point is that it didn't run in a fixed amount of steps. The number of steps required in the algorithm instead depended upon the size of the inputs fed to it. We'll touch on this idea later in the course.

In the 3rd century, Chinese mathematician Liu Hui developed what is currently known as Gaussian elimination (long before Gauss!). Furthermore, he even proved *correctness* of the algorithm (i.e., that the algorithm's instructions conclude with the answer that you would like it to, when used on any input).

In the 9th century, Al-Khwarizmi was part of the Islamic Golden Age, which united ideas from Chinese and Indian number theory in order to develop the number system we currently use. Notably, the word algorithm comes from Al-Khwarizmi himself.

Remark 1.1. The way data is represented is *really* important when performing computation. For instance, you learned how to perform addition when you were 5 or 6 years old using the Arabic numerals, and it wasn't too hard. What if you'd had to learn addition using the Roman numeral system instead? What's MCCXXXIV + MMMMCCCXXI? In Arabic numerals, that's just $1234 + 4321 = 5555$!

In the 19th century, Ada Lovelace produced perhaps the first program, for computing Bernoulli numbers. She is one of the great pioneers of computer science, and the programming language Ada bears her name.

In the 20th century, Alan Turing started contemporary computer science by formally defining computers and algorithms. He also led the team that decoded the Enigma code in order to locate Axis U-boats in the second world war. In 1946, the first programmable electronic computer was created. One last historical note: the first computer bug was a literal bug that got in the hardware of these early computers (hence the name).

What's really the point here?

- Computer science is about much more than programming electronic devices.
- It will improve your **problem-solving skills**.
  - Design, analysis, and implementation of algorithms to solve problems
- It will introduce you to **computational thinking**.
  - Decompose, generalize, abstract, organize
- It will make you a more rigorous and logical thinker.

One last example to really underscore that computers are not (just) electronic devices. One way to solve a famous problem knows as the *Traveling Salesman problem* is by using slime mold! You can literally place food in a petri dish and the slime mold will connect in the shortest path possible.

§1.3 Course information

Here are some of the things you'll learn in the course.

- Problem solving by designing and analyzing algorithms
- Representing and manipulating data
- Programming an electronic computer
  - Using the Python programming language
- Operating an electronic computer
  - Using a terminal on a Linux instance in the cloud

How will you learn all of this?

- Lectures
  - Usually, no slides, live programming and explanations

  - Not mandatory but highly recommended!
  - Lecture notes posted (and lecture hopefully recorded)
- Free textbook: details on Canvas
- Discussions
  - ~10% of final grade, for participation (both mandatory attendance and effort)
  - Make sure you know who your discussion leader is!
  - No swapping discussion sections, sorry
- 9 homework assignments
  - ~35% of final grade
  - Released on Fridays and due the following Friday at 7pm (via Canvas)
  - No homework on midterm weeks
  - -20% for late homework up to 24 hrs past the deadline
- 2 midterms
  - ~30% of final grade, requires a computer
  - Midterm 1: March 4, Midterm 2: April 20
- 1 final project
  - ~25% of final grade; structure subject to change
  - Programming assignment with a partner, project assigned to you
  - 1-2 weeks to complete

There are about 140 students taking this course, so we need to be careful about how you should get help and interact with course staff. Please follow the protocol below.

- **Step 1**: Social office hours:
  - Thursday from 5:30 pm to 9:30 pm, CS Lab 122
  - Friday from 3:00 pm to 6:00 pm, CS Lab 122
- **Step 2**: Email **your own** discussion leader. The email may be forwarded to the head TA if they can't help you.
- **Step 3**: Ask for one on one help with your discussion leader. Again, the email may be forwarded to the head TA.
- Personal matter? Email Professor Jean-Baptiste Tristan.

One last note: if you're going to miss lecture, no need to email anyone. We encourage you to come, but we certainly understand that issues may come up – if you can't come, no need to notify anyone.

Final thoughts: **please read the syllabus**. There's lots of important information, and we were able to cover most, but not all, of it today. Also, there are no discussion sections or office hours this week. Once again, welcome to the course and we'll see you on Friday!

§2 Friday, January 21

§2.1 JupyterHub and primitive types

The very first homework assignment that you'll be completing will be hosted on Jupyter notebooks, which is a flexible platform for writing code, running code, and writing text/math. In particular, a Jupyter notebook is built of different *cells*, that can contain Python code or markdown for writing text.

Within a notebook, you can create new cells, delete existing cells, run cells filled with code and see the output, and write text between cells of code to describe your thought process.

So, for instance, you can have a cell in a notebook that looks like this:

```
2 + 3
```

This is an example of an **expression** in Python, and if you run that cell, it'll output `5`. Awesome! You can also have a cell like this:

```
2 + 2 * 3
```

And this evaluates to 8, as you'd expect. But it's important to note that there was a choice made here – that expression could have instead been evaluated as $(2 + 2) \times 3 = 12$. The fact that it didn't comes down to **precedence** rules; Python has decided that multiplication should be evaluated before addition, which agrees with the way we usually evaluate expressions as humans.

An important fact to note is that every expression in Python has a **type**, which is roughly the 'species' of the expression, or the kind of thing that it is. Important types to start off with are the **primitive types**, which are some of the most basic, built-in types in Python that underlie more sophisticated ideas. For instance, `int` is the type of all the integers $\{\dots, -2, -1, 0, 1, 2, \dots\}$ and `float` is the type that (roughly speaking) contains the continuous real numbers, like 0.26, 1/3, π, etc.[1]

Another import type is `str`, which contains the *strings* in pythons, i.e., collections of characters like `'Hello!'` or `'This is a string :)'`. There are some nice built-in operations on strings, like addition between strings or multiplication of a string by an integer. Let's see that in action:

```
'hello' + ' bob'
```

will evaluate to the string `'hello bob'`, and

```
'hello' * 3
```

will evaluate to the string `'hellohellohello'`. There are tons of built-in operations on these primitive types, and we're simply not going to be able to cover all of them. One important skill that we want to instill in you over the course of the class is the ability

[1] Strictly speaking, computers can't work with all the real numbers precisely, because computers are fundamentally discrete. So in practice it works with mere approximations of these numbers, which can sometimes produce strange behavior. The takeaway is just to be a bit careful anytime you're working with floats, keeping in mind that they're just approximations.

and eagerness to Google your programming questions. It's something that even the pros rely on, and it can be one of the fastest ways to answer your questions.

§2.2 Debugging

Let's talk a bit about **debugging**, which is one of the most important skills in programming. Let's work with an example.

```
2 +
```

If we run this in a cell, we'll get an error that reads `SyntaxError: invalid syntax`. Here's the first, golden rule of debugging: **read the error messages**. They're not *always* useful, but they'll often give you most (or all) of the information that you need to fix the problem.

In this case, we got a `SyntaxError`, which is roughly the programming analogue of making a grammatical mistake. The error will even point you to the line where the mistake was made, and we'd be able to see that we forgot to provide one of the arguments to `+`. Let's look at another example.

```
'hello' ** 5
```

In this case, we would get an error message of a `TypeError`, which tells us that one (or more) of our inputs in an expression has the wrong type. In this case, we would realize that we're not allowed to exponentiate a string and a number (what would it even mean to multiply `'hello'` by itself 5 times?). Let's keep going.

```
4 / 0
```

This gives us a `ZeroDivisionError`, which tells us exactly what we need to know: we tried to divide by 0, which isn't even legal mathematically (much less computationally).

§2.3 Variables

The **variable** is the bread and butter of programmers, and serves as shorthand for the expressions. Let's look at how you **bind** a variable, i.e., assign an expression to it.

```
x = 42
```

This is a line of code that assigns the value 42 to the variable `x`. Unlike in mathematics, it is not declaring that `x` equals 42. After all, `x` doesn't even exist before the line of code is run! But from here on out, we can write code with the variable `x` in place of `42`. To drive the point home, let's look at another (perhaps somewhat surprising) example.

```
x = x + 25
```

This code will run happily! `x` will have the value 67 after the line of code is run, and this underscores that `=` plays the role of an action in Python, not a passive test of equality.

```
b = a + 23
```

What if we run the code above? Well, we've never defined `a`, so Python will yell about a `NameError`, which lets us know that it doesn't know what `a` is. (Good thing we read the error message!)

What if we want Python to display information to us? This is achieved using `print` statements, equipped with an argument of what we want to print. For instance,

```
print(x)
```

will display the value `67` on our screen, as that's the value bound to `x`. Similarly, we can write

```
print('the value of x is:', x)
```

which displays `the value of x is: 67`. (Notice that it added a blank space between the colon and 67.) An important note here is that print statements can be extremely useful for debugging! Riddling your code with print statements lets you know exactly what all the variables are bound to when the code fails, at which step the code fails, etc.

Now here's a bit of a puzzle: what if we wanted our code to print `"hello"`, rather than just `hello`? Running `print("hello")` will achieve the latter, so it's not what we want. It turns out that this can be achieved by combining single quotes and double quotes in Python. If we run

```
print('"hello"')
```

then we'll indeed get `"hello"`, as `print` only strips the outer layer of single quotes.

That should be everything you need for the first homework assignment. In discussion section next week, your TAs will help you familiarize yourself with Jupyter notebooks and the process of transferring homework between Canvas and Jupyter Hub. Good luck on the homework!

§3 Monday, January 24

The goal for today is to learn how to use the terminal; it may not be the most exciting topic we'll cover in the course, but it's important for developing skill in using your own computer in advanced ways and for using computers remotely.

§3.1 Terminal basics

To ease the into idea of using a terminal, recall that last time we talked about using JupyterHub for the first homework. JupyterHub is actually just an interface for accessing a virtual machine in a far-off place, like a warehouse with lots of powerful computers.[2] Furthermore, JupyterHub has its own terminal, accessible from the home page. At a high

[2] Informally, a virtual machine is like a sliver of one of those powerful computers, that you share with many other users.

level, the **terminal** is just a powerful interface for interacting with a computer. The bread and butter of terminal usage lies in its basic commands, some of which we'll cover now (and which we'll expect you to know!).

```
$ echo 'hello'
```

This will have the effect of simply printing `hello` back to us. Nothing too fancy yet. How would we learn more about a terminal command (e.g., about its optional arguments)? Using `man`.

```
$ man echo
```

This will display the manual for echo (hence the name `man`), including lots of information about arguments for echo, etc. Now we have lots of junk on our screen, and we might want to clean things up using the `clear` command.

```
$ clear
```

Now our terminal is clean – nice. In order to get information about the machine that we're using, we can use the `uname` command.

```
$ uname
```

In this case, the terminal will tell us that our machine is using Linux, which can be useful to know.

Now let's talk about commands for organizing data stored in the computer. In order to know where the terminal is currently set up within the file system (it's always somewhere!), you can use the `pwd` command.

```
$ pwd
```

Short for 'print working directory', `pwd` will tell us the **directory** (or folder) where the terminal is currently working. In order to move the working directory, you can use the `cd` command, short for 'change directory.'

```
$ cd ~
```

This will take the terminal to your home directory, since you fed it the `~` argument. You could have written `cd /` to navigate to the root directory instead. (How can you learn more about `cd`? Using `man`!). In order to see the files contained in your current directory, use the `ls` command, i.e.,

```
$ ls
```

If you feed the optional argument `-l` (i.e., write `$ ls -l`), then you'll get even more information about the contents of your current working directory (cwd). You can even make a new directory within your cwd using the `mkdir` command and a name argument, i.e.,

```
$ mkdir my_folder
```

will have the effect of creating a new folder (or directory) in your cwd named `my_folder`. To remove that directory, you would use `rmdir`.

```
$ rmdir my_folder
```

In order to create a file that doesn't exist, say a new text file, you would use `touch`. So

```
$ touch hello.txt
```

will create the file `hello.txt` within your cwd.[3] You can open a file using `open` along with the name of the file, and you can see just the first few or last few lines of the file using `head` or `tail`, respectively. Two last tips for efficiency on the terminal:

1. You can cycle through your previous commands on the terminal using the up arrow; this can save you lots of typing when used correctly!
2. The terminal will auto-complete file and directory names as much as it can when you press tab.

§3.2 Running Python from the terminal

Now let's move on to something a little bit fancier – let's say we've written a Python program in a file called `first.py`. Maybe it looks like this:

```
x = 42
x + 6
```

We can run this from the terminal using the command `$ python first.py`. But nothing happens – why? It's because Python did exactly what we asked; it completed its instructions silently! We can fix this, and ask Python to show us some output, by updating `first.py` as so:

```
x = 42
print(x + 6)
```

Now when we run `$ python first.py`, we indeed see the output of 48. That's an improvement, but it'd be nice to have something more dynamic, perhaps where we can feed the program a number of our choosing at runtime. So we'll update `first.py` again, using the `input` function to request arguments from the user.

[3] If `hello.txt` already exists, then it will just change the 'date last modified' of the file to the current time. That helps explain why it's called `touch` (in fact, using `touch` on a file that doesn't already exist is kind of a degenerate case, even though it may be the most common use).

```
x = input('Give me a number:')
print('Your new number is:', x + 1)
```

Now when we run this, Python actually asks us for input. We can feed it an integer (say 42 again) and look forward to seeing it be incremented by one. But this now gives us a `TypeError` message! Looking more closely, we can see that Python can't compute `x + 1` because `x` is a string.

When Python reads user input via `input()`, it automatically casts it as a `str` type; as humans, we know that we're going to feed the program in integer, but the program itself doesn't know that. So we need to turn `x` into an integer before incrementing it. This leaves us with

```
x = input('Give me a number:')
x = int(x)
print('Your new number is:', x + 1)
```

which will indeed work!

§4 Wednesday, January 26

§4.1 Functions

The goal today is to learn about functions in Python and about approaching algorithmic problems more generally. We'll be running with an example today and for the next couple lectures, concerning solutions of quadratic equations. In particular, recall that a **quadratic equation** is an equation of the form

$$ax^2 + bx + c = 0,$$

where a, b, c are some fixed numbers (with $a \neq 0$) and x is an unknown variable. So a particular quadratic equation might look something like $3x^2 + 2x + 1 = 0$. The name of the game is to find the value(s) of x that make this equation hold true, known as **roots**.

From a mathematical perspective, this problem has been resolved using the quadratic formula, which we'll discuss in more detail shortly. From a computer science perspective, however, there's a bit more going on. The precise problem setup is that there are 3 inputs – the numbers a, b, c (with $a \neq 0$) – and the desired output is a pair of numbers x_1, x_2 such that

$$\begin{aligned} ax_1^2 + bx_1 + c = 0, \\ ax_2^2 + bx_2 + c = 0. \end{aligned}$$

Now the goal is to find a generic procedure that works to send *any* valid input values a, b, c to (correct) output values x_1, x_2. As we've discussed previously, this kind of generic procedure for computing output from input is known as an **algorithm**.

Now back to the quadratic formula; here's the precise mathematical statement.

Lemma 4.1 (Quadratic formula)
Let $f(x) = ax^2 + bx + c$ be a quadratic equation (i.e., $a \neq 0$). Then the (complex) roots of f are exactly

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

A crucial skill for any computer scientist, which is a bit hard to teach, is how to turn a mathematical result (like the quadratic formula) into computer code. So let's try to start by at least computing the roots of a particular quadratic equation, say $x^2 + 4x + 2$. We might start writing code as follows.

```
a = 1
b = 4
c = 2
```

We have an idea of how to perform the addition, multiplication, and division necessary to compute the result of the quadratic formula, but how are we going to calculate the square root? After all, there's no built-in square root character in Python like `+` for addition or `*` for multiplication.

This is where we'll turn to Python's **libraries**, which are collections of functions that other people have written for us. In this case, we'll use the `math` library, which has the function `math.sqrt()` that we're looking for.[4] Now we can write our code as follows.

```
x1 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
x2 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
```

Awesome! Now what if we want the roots of another quadratic equation? We'll need to update the values of `a`, `b`, and `c` and run all these lines of code again. That's not terribly inconvenient in this case, but it would be really inconvenient for larger suites of code, and we can do better. The technique for packaging many lines of code into a single name in Python is the **function**. Before trying to write a function that solves quadratic equations, let's get our feet wet with simple functions.

```
1  def my_func(x):
2      x = x + 1
3      print('Value of x:', x)
4
5  my func(5)
```

In the first three lines, we're defining our own function `my_func` with the single **parameter** `x` that increments `x` by 1 and prints it. In the fifth line, we're actually using `my_func` with the **argument** 5, which will result in Python printing `Value of x: 6`. Simple enough. Now here's a bit of a puzzle – what if we ran the following code?

[4] If you were to forget the name of a library or a function within a library (and everybody does), remember to use Google! Even the pros do it.

```
1 x = 42
2 def my_func(x):
3     x = x + 1
4     print('Value of x:', x)
5
6 my func(5)
```

We'll actually still get Python printing `Value of x: 6`. So the first line, in which we wrote `x = 42`, didn't affect the computation of `my_func(5)` at all! The fact that `my_func` doesn't care about the value of `x` outside of its own argument/definition has to do with the idea of **scope**. In particular, the value of `x` on line 3 only comes from the argument to `my_func`, not from anywhere else.

Now what if we were to run `my_func(4+5)`? This would be evaluate as `my_func(9)`, due to the **call by value** nature of Python. In particular, Python first evaluates `4+5` and then sends the result to `my_func`. That's how most programming languages do things, but not all of them![5]

Okay, back to the quadratic equation. Let's try to write a solver for it now that we're warmed up.

```
def solver(a, b, c):
    x1 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    x2 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

That works! If we run

```
solver(1, 4, 2)
```

then we'll get the roots of $x^2 + 4x + 2$ that we saw earlier. If we want the roots of $x^2 + 5x + 2$, then we only need to change a single character.

```
solver(1, 5, 2)
```

We've made a great leap forward from just writing code in cells (i.e., without functions), but there are still improvements to be made. The code is currently a little bit hard to read and has redundant computation (e.g., we're computing the square root of `b**2 - 4*a*c` two times). Let's try to clean this up a bit, with the goal of making it faster, easier to read, and less prone to errors.

```
def solver(a, b, c):
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
```

[5] If this is confusing, don't worry. The idea is really just that Python does what you'd expect it to do in this situation, and some other languages do stranger, fancier things.

```
    print('First solution:', x1)
    print('Second solution:', x2)
```

This code still isn't perfect (we'll see why over the next several lectures), but it's faster and cleaner than the previous version of `solver()`. Now let's conclude with a bit of debugging. What if we were to run:

```
solver(1, 2)
```

Then Python will give us an error message:

```
TypeError: solver() missing 1 required position argument: ‘c’.
```

That's pretty descriptive; we now know that we need to feed `solver` its third argument. What if we were instead to go overboard with arguments?

```
solver(1, 2, 3, 4)
```

We get another error message:

```
TypeError: solver() takes 3 positional arguments but 4 were given.
```

Again, that's pretty informative and helps us figure out what went wrong. Now let's stop messing around and really use `solver()` as intended.

```
solver(1, 4, 2)
```

This time we've made sure to give `solver()` the correct number of arguments of the correct type. But we still get an error! Python will complain of:

```
ValueError: math domain error.
```

In this case, what's happened is that `math.sqrt()` is trying to take the square root of a negative number! We should have been more clever in writing `solver()` and made sure that it didn't accept arguments causing this kind of problem. We'll see how to approach these kinds of issues in the coming lectures.

§5 Friday, January 28

§5.1 Announcements

We'll keep talking about functions today, and you should be ready to start tackling Homework 2 by the end of the lecture. Speaking of the homework, we gave a quick look at some of the Homework 1 submissions yesterday, and it looks like the following thing is happening: many of you have already done some programming before, yet you answer a homework problem incorrectly and (in addition) use unnecessarily advanced techniques.

So, please, read the questions carefully before writing your solutions and be cautious about using sophisticated techniques when you really don't need to.

One more thing to mention: there's a collegiate programming contest called the ICPC, which BC used to perform very well in but which we don't seem to participate in anymore.

Professor Tristan is going to try to restart BC's participation in this competition, so please let him know if you're interested. To sweeten the deal, keep in mind that the kind of algorithmic thinking needed for these competitions is great preparation for interviews at places like Google, quantitative finance companies, etc. (As we mentioned earlier, algorithmic thinking is a very difficult and valuable skill to develop!)

§5.2 Problem: sum of roots

Last time we wrote functions for finding roots of quadratic equations. Today, we'll change things up a bit and think about writing a function that returns the sum of a quadratic equation's roots. That is,

$$a, b, c \longmapsto x \text{ such that } \begin{cases} x = x_1 + x_2; \\ ax_1^2 + bx_1 + c = 0; \\ ax_2^2 + bx_2 + c = 0. \end{cases}$$

Now, there are a couple of ways to approach this problem. Perhaps the most obvious is to use our function `solver()` from last time in order to compute x_1 and x_2, and then return their sum. Breaking this down further, there are two ways to implement this idea:

1. Copy and paste the body of our `solver()` function into a new function that prints $x_1 + x_2$ instead of x_1 and x_2 separately.

2. Write a function that uses the output of `solver()` in order to compute $x = x_1 + x_2$.

Path (2) is far, far better than path (1). **If there is a golden rule of programming, it is to not copy and paste code**. When you copy and paste code 5 times, then any bug you find needs to be fixed 5 times, any style change/clean-up you make needs to be implemented 5 times, any efficiency speed-up needs to be implemented 5 times, etc. Also, copy-pasted code is much harder to read than modular code.

Now, in order to implement idea (2), we need to change `solver()` so that it actually returns its output, rather than just printing it. This is achieved using the `return` keyword. Our new `solver()` will be as follows.

```
1 def solver(a, b, c):
2     sr = math.sqrt(b**2 - 4 * a * c)
3     x1 = (-b + sr) / (2 * a)
4     x2 = (-b - sr) / (2 * a)
5     return x1, x2
```

§5.2.1 Tuples

Something a bit subtle is happening in line 5; `solver()` is returning two values at once by using the `tuple` type. In contrast to the **primitive types** that we have seen previously, the `tuple` is a **composite type**, meaning it's a bit more complicated and built from the simpler primitive types. At a high level, the `tuple` is simply a type for placing several items *in order*. For instance,

```
x = (2, 3)
```

binds `x` to the tuple with the `int` 2 in the first position and 3 in the second position. If you've seen vectors before, you can think of tuples like that.

In fact, tuples are one kind of a **data structure**, which is a format for storing and manipulating data. If this is a bit confusing, don't worry – we'll encounter several more data structures over the course of the class. In fact, data structures are probably the second most important idea in computer science, after algorithms!

Now we need to familiarize ourselves with tuples a bit. Let's say we have `x = (2, 3)` as before – how can we access the entries 2 and 3 from the variable x? This is achieved by **indexing** into x, via the following syntax.

```
two = x[0]
three = x[1]
```

By running this code, the variable `two` will indeed take the value 2 (corresponding the leftmost entry of x), while `three` will take the value 3 (corresponding to the rightmost entry of x). An important observation here is that **0-indexing** is used when accessing the entries of x. That is, the leftmost entry of x has the index 0, while the next one has the index 1, and so on. Simply put, we start off counting from 0 rather than 1.

This is just a convention in computer science. Python could have chosen to start indexing tuples by starting with the number 43, and that would be perfectly legal (though very confusing for humans). For whatever reason, computer scientists often like to start counting at 0. (Kind of like how in Europe, the ground floor of a building is the 0th floor, rather than the 1st.)

Another way to grab the entries of a tuple is via **pattern matching**, as follows.

```
two, three = x
```

This will again have the effect of binding `two` to 2 and `three` to 3. So now that we've learned about tuples, we can write the outer function that uses `solver()` and adds its output (in order to solve our original problem about sums of roots). We can write:

```
def solver_sum(a, b, c):
    x1, x2 = solver(a, b, c)
    print(x1 + x2)
```

Isn't that nice? We solved our new problem with two lines of code! This is just an example of the power of writing modular code, i.e., code that is split up into several functions that can be reused, rather than huge blocks of copy-pasted code.

§5.3 More operations

Now we're going to learn about more of Python's powers. We just learned about indexing into tuples, which are entries of data in order. That doesn't sound so different from strings (which are just characters in order), so maybe we can index into them as well. Let's try.

```
s = 'Hello, I am Sam!'
s[4]
```

This indeed returns `'o'` – nice! What if we want all the values between two indices of the string? We can do so via **slicing**, i.e.,

```
s[3:12]
```

This returns `'o, I am '`, the values between the 3rd and 12th indices. To get the string's entries from the 3rd index onward, you can write `s[3:]`, and to get the string's entries up until the 7th entry, you can write `s[:7]`.

Now let's move from strings to floats; what if we want to round a `float` to an `int`? We can use the built-in `round()` function, that rounds a `float` to the *nearest* `int`. For instance,

```
round(4.8)
```

returns 5, while `round(4.2)` returns 4. What if we want to round to the next-lowest integer or next-highest integer, rather than the closest? Then we'll need to use the `math` package, with the functions `math.floor()` or `math.ceil()` respectively. For instance,

```
import math
math.ceil(4.1)
```

comes out to 5.

§6 Monday, January 31

§6.1 HW1 postmortem

Quick comment on the homework due last Friday: the goal was to compute the following value, as a function of T and v:

$$T_{wc} = 35.74 + 0.625 \cdot T - 35.75 \cdot v^{0.16} + 0.4275 \cdot T \cdot v^{0.16}.$$

Most of you wrote something like this:

```
1  T = 20
2  v = 15
3  35.74 + 0.625 * T - 35.75 * v ** 0.16 + 0.4275 * T * v ** 0.16
```

That will produce the correct value, but it actually has an imperfection. Namely, the value `v**0.16` will be computed *twice* by Python in the third line. Rather than having Python repeat identical computation, and waste time & energy, we can do better by introducing a variable that stores the value of `v**0.16`.

The intended solution was as follows.

```
T = 20
v = 15
tmp = v**0.16
35.74 + 0.625 * T - 35.75 * tmp + 0.4275 * T * tmp
```

In particular, the variable `tmp` (for temporary) introduced in the third line saves Python from repeating computation in line 4.

One more note: Python evaluates code in a line-by-line manner, and it doesn't evaluate the body of a function until the function is actually called with arguments. For instance, consider the following code.

```
1  x = 10
2  y = x + 42
3
4  def test(x):
5      print('hello')
6      return x + 1
7
8  test(y)
```

What will this output? Well, lines 1 and 2 are run in that order, so `x` = 10 and `y = 52`. Then `test(x)` is defined but its body is not run (since we haven't called it with any input yet!). In line 8, finally, we call `test(y)`. Since `y` is bound to 52, that comes out to `test(52)`.

So this code will print `'hello'` and line 8 will return the value 53. In particular, that's exactly the same as if we had replaced lines 1 and 2 with the single line `y = 52`. The body of `test()` only cares about the `x` that it is given as an argument.

§6.2 Booleans

The primary goal for today is to introduce a new type along with its primary functionalities. In particular, we will introduce the type **bool** of Boolean values (named after mathematician George Boole). The type `bool` has only 2 values: `True` and `False`. They're really meant to express the usual notions of truthhood and falsehood within Python, and to allow us to branch our computation based on whether something is true (i.e., compute `f(x)` if `P(x)` is true and `g(x)` otherwise).

Let's jump into some examples, demonstrating how we can get `bool`s from types we already know. For instance,

```
2 < 3
```

will evaluate to `True`, just as

```
3 <= 3
```

will evaluate to `True`. On the other hand,

```
2 > 3
```

and

```
3 <= 4
```

will both evaluate to `False`. So `<` and `<=` correspond to testing strict and weak inequalities of numbers.

To test for equality, among numbers and various other types, you can use `==`. (Recall that `=` is already taken for variable assignment, rather than testing equality!). So,

```
3 == 4
```

and

```
'hello' == 'goodbye'
```

will evaluate to `False`, whereas

```
5 == 5
```

evaluates to `True`.

So that's how we can get `bool`s from familiar types. But we can also manipulate `bool`s themselves to get other `bool`s. For instance, `not` applied to a single Boolean `x` will evaluate to `False` if `x` is `True` and to `True` if `x` is `False`.[6]

So, combining what we've learned,

```
not True == False
```

will itself evaluate to `True`! Two keywords for combining two Boolean expressions (rather than a single one) are `and` and `or`. Once again, they're built so as to agree with their plain English names. So,

```
True and True
```

comes out to `True`, while

```
True and False
```

comes out to `False`. On the other hand, `or` of several expressions will evaluate to `True` as long as even a single one of the argument expressions is `True`. For instance,

```
True or False
```

is `True`, and

```
True or True
```

is `True` as well.

[6] Nothing too fancy here; these Python keywords are designed so as to agree with plain English.

§6.3 Conditional statements

Now let's get to the most important use of `bool`s: **branching** computation. In particular, you often want your program to compute `A(x)` if `P(X)` is true and `B(x)` if `P(x)` is false.

The syntax for this lies in the keywords `if` and `else`. Let's learn through example.

```
1 x = 3
2
3 if x == 3:
4     print('x is 3 and I executed the top branch!')
5 else:
6     print('x is not 3 and I executed the bottom branch!')
```

The rule here is the following: if `x == 3` evaluates to `True`, then running this code will run the indented code immediately after `if` and skip the code after `else`. If `x == 3` evaluates to `False`, then running the code will skip the indented code after `if` and run the code after `else`. So, in this case, we'll see this message printed: `'x is 3 and I executed the top branch!'`. If we had set `x = 4` on line 1, then running this code would result in (only) the other message being printed.

You can also branch on more than one condition by using the `elif` keyword, which is short for `else if`. So,

```
if x > 1:
    print('Took first branch')
elif x > 2:
    print('Took second branch')
else:
    print('Took third branch')
```

will send 1.5 to the first branch (and nowhere else), 2.5 also to the first branch (and nowhere else), and 0.5 to the third branch (and nowhere else). In fact, no numerical value of `x` will reach the second branch – can you see why?

Now let's rewind to when we were writing our `solver()` function for finding quadratic roots. Recall that it looked like this.

```
1 def solver(a, b, c):
2     sr = math.sqrt(b**2 - 4 * a * c)
3     x1 = (-b + sr) / (2 * a)
4     x2 = (-b - sr) / (2 * a)
5     print('First solution:', x1)
6     print('Second solution:', x2)
```

One problem we ran into earlier is that `math.sqrt()` doesn't accept negative input, which can sometimes happen in line 2 if we allow for any inputs `a, b, c`. We also want to make sure, as a basic first step, that $a \neq 0$ (otherwise, lines 3 and 4 really don't make sense . . .). We can improve this function a bit using our new knowledge of conditional statements.

```
def solver(a, b, c):
    if a == 0:
        print('a should be 0!')
        return
    tmp = b**2 - 4 * a * c
    if tmp < 0:
        print('No real solutions')
        return
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

Now `solver` will return nothing and print a disclaimer when it gets problematic input. Nice. Another problem we had was that `solver()` accepts input from types other than `float` and `int`.

We can get the type an expression using `type()` and test that it equals `int` or `float` using `==`. For instance,

```
type(3) == int
```

comes out to `True`, while

```
type('hello') == float
```

comes out to `False`. Next time, we'll see how to use this kind of technique to improve `solver()` even further by having it only accept numbers.

§7 Wednesday, February 2

A quick errata from last time: we were talking about the implementation of ordering on strings, e.g., how

```
'be' < 'be curious'
```

evaluates to `True`. We thought that this tested whether the left hand side is a substring of the right hand side, but that's actually not true. `<` instead compares strings under the dictionary ordering (i.e., the ordering used to list words in the dictionary, or to list your last names on Canvas).

§7.1 Type checking

Last time we started talking about the need to check the types of arguments given to the functions that we write. For instance, to make sure that `solver()` only takes `int`s and `float`s as input, we can write a helper function `ct()` (for check type).

```python
def ct(x):
    return type(x) == int or type(x) == float
```

So `ct()` returns `True` if `x` has the right type for `solver()` and `False` otherwise. Let's remind ourselves of the version of `solver()` that we were looking at last time, and think about how to use our new helper function `ct()` .

```python
def solver(a, b, c):
    if a == 0:
        print('a should be 0!')
        return
    tmp = b**2 - 4 * a * c
    if tmp < 0:
        print('No real solutions')
        return
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

We're currently checking to make sure `a` does not equal zero, and that the polynomial indeed has real roots (rather than complex roots), which is great. It would be even better to kick things off by checking the types of `a` , `b` , and `c` . Making use of `ct()` , our new `solver()` will look as so:

```python
def solver(a, b, c):
    if not (ct(a) and ct(b) and ct(c)):
        print('One argument has the wrong type!')
        return
    if a == 0:
        print('a should be 0!')
        return
    tmp = b**2 - 4 * a * c
    if tmp < 0:
        print('No real solutions')
        return
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

And indeed we can check that `solver(1, 2, 'hello')` returns nothing and prints our new error message.

Remark 7.1. We're not doing anything terribly fancy, but note that this version of `solver()` uses all the techniques we've learned about: functions, conditional statements, and type checking!

Now we've made sure that `solver()` doesn't return anything for illegal inputs, but it still runs perfectly well, which isn't ideal. If we were in a larger project with thousands of lines of code, and `solver()` were being used somewhere deep in a complicated process, then our current setup would have some serious drawbacks:

- `solver()` can feed incorrect output (i.e., nothing) to later functions that happily make use of it.
- We won't know the line number or even the function where we used arguments of the wrong type.
- Our entire program (in which `solver()` is just one tiny piece) will still run, even when *we know* there's a mistake.

The way we can fix all these issues is by **raising an error**, which is Python's way of halting a program, spitting out an error message, and pointing the user to the function and line number where the error occurred. (All of these functionalities are extremely useful!). We can rewrite `solver()` to raise errors like so.

```python
def solver(a, b, c):
    if not (ct(a) and ct(b) and ct(c)):
        raise ValueError('One argument has the wrong type!')
    if a == 0:
        raise ValueError('a should be 0!')
    tmp = b**2 - 4 * a * c
    if tmp < 0:
        raise ValueError('No real solutions')
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

§7.2 JupyterHub

This Friday (next class!) we'll have a brief programming quiz to help prepare you for the structure of the midterm. The quiz this Friday won't be graded, but it's still a good exercise to make sure you're on track, and you should make sure that you're comfortable working with JupyterHub beforehand (e.g., creating Python files, writing basic Python files, running Python files from the terminal).

So, let's say we're at your terminal in JupyterHub. If our goal is to run a script that prints `'Hello'`, then we would proceed as so.

```
$ touch hello.py
```

Now we've created the new Python file `hello.py` (note that its name needed to end in `.py`!). Then, in `hello.py`, we can write:

```
print('Hello')
print('Goodbye')
```

Now, *after saving the code we wrote in* `hello.py`, we can run the program in the terminal.

```
$ python hello.py
```

And this will indeed print `'Hello'` and `'Goodbye'`. Let's update our code to take in the user's name.

```
print('Hello, my name is HAL')
s = input("What's your name?")
print('Nice to meet you', s)
```

Upon saving `hello.py` and running it from the terminal, we get the desired behavior.

```
$ python hello.py
Hello, my name is HAL
What's your name? John
Nice to meet you, John
```

Cool. It's a little bit annoying to have Python ask the user for values one-by-one, so let's see how can do things a bit more efficiently.

Let's first make a new file.

```
$ touch test.txt
```

And fill it like so.

```
John
```

Now we can use the lines of `test.txt` as input to `hello.py`! The syntax is as so:

```
$ python hello.py < test.txt
Hello, my name is HAL
What's your name?
Nice to meet you, John
```

With this terminal command, `hello.py` will be run, and it will take successive lines of `text.txt` as input any time it requests input. So the second line of `hello.py`, that requests the user's name, went ahead and grabbed the first line from `text.txt`.Make sure you understand this example, because something similar will show up on Friday's quiz!

§7.3 Problem: maximum of 3 integers

Now we're going to discuss a new problem: given three integers a, b, and c (appearing on their own lines in a .txt file), write a program that prints the largest of the 3 integers. Here's a high-level tip on how to approach these kinds of problems: **Start by thinking about the problem with pen and paper, and try to sketch a solution. *Then* try to code up the idea you developed on paper.**

So let's start by thinking about our problem in English (and *not* jump into writing code!). So, one way to cast this problem mathematically looks like this:

$$(a, b, c) \mapsto \begin{cases} a & \text{if } a > b \text{ and } a > c, \\ b & \text{if } b > a \text{ and } b > c, \\ c & \text{if } c > a \text{ and } c > b. \end{cases}$$

We can code that up as follows, in `solution.py`.

```python
1  a = input()
2  b = input()
3  c = input()
4
5  if a > b and a > c:
6      print(a)
7  elif b > a and b > c:
8      print(a)
9  elif c > a and c > b:
10     print(c)
```

Now if we run this on inputs `23, 265, 45`, we get the output 45. What? What went wrong here? In lines 1, 2, and 3, Python takes in its input as `str` types, not `int`s. So it is comparing `a`, `b`, and `c` as strings under the dictionary ordering we mentioned earlier! Let's modify `solution.py` and fix this.

```python
1  a = int(input())
2  b = int(input())
3  c = int(input())
4
5  if a > b and a > c:
6      print(a)
7  elif b > a and b > c:
8      print(a)
9  elif c > a and c > b:
10     print(c)
```

One important skill for you to learn is to be adversarial when thinking about the code you write. Imagine that someone were to pay you 100$ if you could break your code – what would you do? Always try to think about edge cases that can break your code, for instance. In this case, what if we were to run `solution.py` with inputs `100`, `100`, and `100`?

Uh oh, running this script with `100`, `100`, `100` gives us no output at all. What went wrong? In this case, it turns out our math was wrong! The work we did with 'pen and paper' (or on the blackboard, in this case) was totally bogus. The mathematical formulation we wrote above is incorrect, and fails when there is a tie for the largest number. So the code we wrote based on our math was busted as well.

We can fix `solution.py` by using weak inequalities, rather than strict ones.

```python
a = int(input())
b = int(input())
c = int(input())

if a >= b and a >= c:
    print(a)
elif b >= a and b >= c:
    print(a)
elif c >= a and c >= b:
    print(c)
```

This code is now correct, but it's also fairly inefficient. It makes 6 comparisons in order to find the maximum of 3 numbers, which is quite high. (Finding such a maximum can be done with only 2 comparisons!). Next time we'll see how to improve upon this, and we'll be talking much more about the efficiency of our programs later on in the course.

§8 Friday, February 4

We started things off with a brief (ungraded) coding assessment, as we mentioned last time. It's an important exercise in writing a program in Jupyter, downloading it to your computer, and uploading it to Canvas. We'll do something similar next week, and it's really important that you get familiar with this process before the midterm comes around!

§8.1 Problem: maximum of 3 integers (continued)

So, last time we were talking about the problem of finding the maximum of 3 integers. Remember what the process of solving this problem should look like:

1. Read the problem carefully and understand it.
2. Think of an algorithm for solving it, via pen and paper.
3. Code up your algorithm.

Last time, we developed the above program for solving this problem. We can break it down into a more modular and readable form as follows:

```python
def get_input():
    a = int(input())
    b = int(input())
    c = int(input())
    return (a, b, c)

```

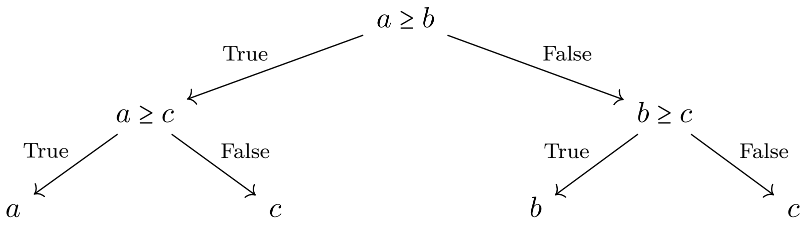
```python

def solution(a, b, c):
    if a >= b and a >= c:
        print(a)
    elif b >= a and b >= c:
        print(b)
    elif c >= a and c >= b:
        print(c)

(a, b, c) = get_input()
solution(a, b, c)
```

This solution is correct, which is great, but we there's still something pretty unpleasant about it – it can take up to 6 comparison to find the maximum of 3 numbers. That's a lot. Thinking about it informally, it seems like we should be able to get away with far fewer.

In particular, think about the following graph for comparing pairs in a, b, c and making deductions.



We've reformulated the mathematical thinking underlying a potential solution, but what's really the point? The key idea is that this new framing requires only two comparisons, regardless of the values of a, b or c! That's a considerable improvement over the 6 comparisons in our old algorithm, and it's the start of our thinking on the *efficiency* of algorithms.

At a high level, any given problem will have many potential solutions (i.e., programs that always output the right answer). But some are far better than others, as measured by how fast they run (i.e., the number of operations they use). This field – of measuring how efficient algorithms are, and how many steps they take – is known as **analysis of algorithms**. All analysis of algorithms is really just a counting game, i.e., counting how many times an algorithm needs to perform a basic operation.

We'll discuss this in more depth later on in the course, but for now the key idea is that we've found another solution to our problem that's equally correct (i.e., produces the right output) but superior in efficiency. Now let's code up the diagram we drew earlier.

```python
def get_input():
    a = int(input())
    b = int(input())
    c = int(input())
    return (a, b, c)



def solution(a, b, c):
    if a >= b:
        if a >= c:
            print(a)
        else:
            print(c)
    else:
        if b >= c:
            print(b)
        else:
            print(c)

(a, b, c) = get_input()
solution(a, b, c)
```

Awesome. We can save this in the file `faster.py` and write three input lines in a file `test.txt` to test it out. Say `test.txt` has the lines:

```
23
45
1234
```

We run `faster.py` with the arguments in `text.txt` the same way we did last time, in the terminal like so.

```
$ python faster.py < text.txt
1234
```

And this indeed gives us the output of `1234`, as we hoped it would.

Remark 8.1. Warning: on a midterm or quiz, **do not** add any strings to `input()` to politely request the input, unless we ask you to! For instance, if you instead write `input('Please give me a number:')`, then your program will be printing that message when it gets run, when it should only be printing the answer to the problem.

§9 Monday, February 7

Two things to mention:

1. It looks like many of you started last week's homework quite late, maybe a couple of hours before the due date. This might work for the first or second homework, but it's definitely not going to work for homeworks later in the course. Please keep in mind that things are going to ramp up.

2. Few of you succeeded in the ungraded quiz last Friday. For this reason, we're going to do it again this Friday. As we've already mentioned, it's really important to have this process down before the midterm.

§9.1 Iteration

Today we'll be talking about iteration, which is actually one of the last programming techniques we'll be talking about in the course. Once you add this to your toolkit, you'll be able to convert almost any algorithm from English to Python.

Let's start things off with an example. The problem is as follows: given n, compute the number of integers in $\{1, 2, \ldots, n-1, n\}$ that have 3 as a factor but do not have 11 as a factor. Perhaps the most obvious solution is to simply check each number less than or equal to n and see if it is divisible by 3 but not 11.

It would be pretty hard to code this up (for arbitrary n!) using only the techniques we already know. Fortunately, a new technique known as the **while loop** allows us to implement this idea fairly easily. At a high level, a `while` loop will:

1. Check whether a condition is true.
   a. If true, run the body of the `while` loop (i.e., the indented code immediately after the `while`) and return to step 1.
   b. If false, escape the `while` loop and move on.

Here's an example.

```
1 while 3 <= 5:
2     print('Hello')
```

This program will just print `'Hello'` forever! Python first checks whether `3 <= 5`. That's true, so it executes the second line (prints the message) and returns to line 1. Line 1 evaluates to `True` again, and the cycle continues.

This isn't too useful an application of `while`; we're better off making sure that it checks a condition that eventually evaluates to false. Let's try to make that happen.

```
1 i = 0
2 while i <= 12:
3     print('Hello')
```

Hm, this still goes on forever. The reason why is that `i` never changes from `0`, so line 2 will always evaluate to `True`! Let's try this again, making sure that `i` gets bigger as we go.

```
i = 0
while i <= 12:
    print('Hello')
    i = i + 1
```

This time we only get `'Hello'` 13 times - nice! Notice that the body of the `while` loop (i.e., the indented code following the loop) indeed should get called 13 times: once for $i = 0, 1, \ldots, 12$ (since `12 <= 12` is `True`).

Let's get even more information:

```
1 i = 0
2 while i <= 12:
3     print('Hello', i)
4     i = i + 1
```

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10
Hello 11
Hello 12
```

So the loop is indeed doing what we think it is: checking the condition in line 2, running lines 3 and 4 if true, checking the condition in line 2 again, running lines 3 and 4 if true, etc. Let's look at an edge case now.

```
1 i = 0
2 while i <= -1:
3     print('Hello', i)
4     i = i + 1
```

What do you think this will print? Well, line 2 checks whether `i <= -1`, i.e., whether `0 <= -1`. That evaluates to `False`, so the programs skips all the indented code immediately after `while`, which is the rest of the program. So this program prints nothing!

§9.2 Problem: multiple of 3 but not 11

Now let's play around in JupyterHub, and start off by making a small workspace for ourselves.

```
$ mkdir LectureExamples
$ cd LectureExamples
$ mkdir Morning
$ mkdir Afternoon
$ cd Afternoon
$ touch problem1.py
$ touch test.txt
```

Now we have some nice folders and a Python file in which to write our solution to the 'multiple of 3 but not 11' problem. Let's now write our solution in the form of a function, which is generally good practice. In `problem1.py`, we'll write the following.

```
 1 def get_input():
 2     n = input()
 3     m = int(n)
 4     return m
 5 
 6 def solution(n):
 7     if n < 0:
 8         raise ValueError('Your value is too small.')
 9 
10     i = 0
11     count = 0
12 
13     while i <= n:
14         if i % 3 == 0 and not i % 11 == 0:
15             count += 1
16         i += 1
17     return count
18 
19 a = get_input()
20 answer = solution(a)
21 print(answer)
```

Think carefully about what we've written in this solution, and make sure you understand it. A very important note: **indentation is extremely important in Python**. If we were to indent line 16, for instance, we would have a completely different program!

§10 Wednesday, February 9

§10.1 HW2 postmortem

Before we keep talking about loops, let's say a few things about the homework: we know some of you were surprised about your score on the previous homework, so it's worth having a brief discussion about it.

First things first: we're going to be changing the homework a bit moving forward, so as to have leaner programming questions without much dialogue/story surrounding them.

Now here's a bit of a rhetorical point: imagine you want to build a house from scratch, and the house will have three floors. You hire an architect and an engineer, and together they build the house. Then the house actually collapses before it can be delivered to you. They tell you that the second and third floors were built perfectly, but the first floor was built imperfectly and thus the house collapsed. Did they do a good job? How would you measure their performance?

A similar phenomenon happened on the homework for some of you – we know that's frustrating, and this kind of dependence won't happen much on future homework, but maybe this example helps you understand how we graded the homework.

One more thing to talk about: some of you used super-powered techniques to solve the homework problems, like dictionaries, splitting a string with a separator, etc. You'll get full points on that for now, because we're using an automated grader, but this is really hindering you from learning how to be creative and resourceful with Python.

Every homework can (and should!) be solved using only the techniques we learned *before* it was assigned. Coding under this kind of restriction will make you a more clever programmer, and we'll get to many of the fancier data structures later in the course anyway (at which point you'll really be forced to be clever in the homework!).

§10.2 Arrays

So, we've learned about a handful of primitive types: `int`, `float`, `bool`, and `str`. We've also learned about the data structure `tuple`, which stores several values in order. For instance, `x = (2, 3)` will set `x` to have the type `tuple`. Now what if we change our mind about the values in `x` and want it store 4 in the second entry, for instance? Easy peasy, we just use the syntax we already know for **indexing** into a tuple (i.e., accessing its elements via their positions).

```
x[0] = 4
```

Uh oh, this gives us a `TypeError`: `'tuple' object does not support item assignment`. Simply put, once you create a tuple, its values are set in stone forever – you're not allowed to change them. The word for this kind of property, where you can't change a data structure's content, is **immutable**.

In order to overcome this limitation (certainly *some* of the time we'll want to have a data structure whose entries we can change), we'll need to learn about a new data structure. Namely, the **array** type comes to the rescue. It is a data structure for holding entries of data in order and which is **mutable**, i.e., whose contents can be changed after they are created.

Let's dive into things: how do we actually create an array? We can create an array of size 10 with a 0 in each entry like so.

```
arr = [0] * 10
print(arr)
```

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

So the variable `arr` is now bound to our array. How can we think of this array? Roughly speaking, you can think of it as a chunk of 10 contiguous blocks of memory in your computer. Currently, each block of memory has the value 0, but we can change the value at the 5th index like so.

```
arr[5] = 12
print(arr)
```

```
[0, 0, 0, 0, 0, 12, 0, 0, 0, 0]
```

We can retrieve the values in our array (or *index* into our array) via similar syntax.

```
arr[4]
```

```
0
```

Now let's look at the type of this thing.

```python
type(arr)
```

```
list
```

There's a bit of a notational inconvenience here: strictly speaking, Python only supports lists, which are slightly different than arrays, but we'll be thinking of them as arrays in this course (and using lists only as arrays will make your code run fast!). Another important operation to know is that you can retrieve the length of your list via `len(arr)`, which comes out to 10 in this case.

There's only one more thing we want you to know about for arrays, which is how to sort them. Simply put, sorting an array just means shuffling around its elements so that they are in ascending order. This can be done in Python as so.

```python
arr = sorted(arr)
```

Sorting is a really powerful tool that can often render a very difficult problem into a trivial one. Now you know each of the 5 tools we expect you to know about arrays:

1. Creation (`arr = [3] * 20`)
2. Assignment (`arr[12] = 25`)
3. Indexing (`x = arr[3]`)
4. Length (`n = len(arr[12])`)
5. Sorting (`arr = sorted(arr)`)

§10.3 For loops

Now we'll be learning about a new kind of loop, which you can think of roughly as a cousin to the `while` loop. Simply put, the **for loop** is like a `while` loop in which the indexing of a counter is handled for you. For instance, `while` loops (especially those that interact with arrays) will often look something like:

```python
i = 0
while i < len(arr):
    # do cool stuff
    i += 1
```

That's perfectly fine, but it's very easy to forget to increment `i` within the body of the `while` loop, especially as your loops get fancier and more complex. The `for` loop can be thought of as a `while` loop that has this incrementing automatically baked into it. For instance, the previous `while` loop is exactly equivalent to the following `for` loop.

```python
for i in range(len(arr)):
    # do cool stuff
```

In particular, this `for` loop executes its body (in this case just `# do cool stuff`) while i takes the values `0, 1, ..., len(arr) - 1`. For instance, the following loop will just print `0, 1, ..., 8`.

```
for i in range(9):
    print(i)
```

That's everything for today – on Friday we'll do another ungraded coding quiz, as we mentioned last time.

§11 Friday, February 11

We kicked things off with another ungraded coding quiz, as we prefaced earlier this week. An important homework note: from now on, the homework will be distributed to you via JupyterHub in the directory `CS1_2022`. All the problem statements are located in that directory, and you can also look at them on github's website https://github.com/jtristan/CS1_2022. We even provide some testing code so that you can be sure that your solution is reading input and outputting solutions correctly.

§11.1 Problem: find x, y, z

Here's a problem for us to warm up: let x, y, and z be 3 positive integers such that $x \leq y \leq z$. You are given 7 integers: x, y, z, $x+y$, $x+z$, $y+z$, $x+y+z$, in some unknown order. Determine the value of x, y, and z.

First (before writing any code!) take a couple minutes to think about how to solve this problem. Think of a rough skeleton of the program you would write. How do you know it's correct? Is it efficient?

Now let's get our hands dirty and actually build the solution. We're going to be receiving these 7 integers one by one, e.g., via

```
$ python soln.py < input.txt
```

where `input.txt` has those 7 integers on separate lines. So we'll need to use `input()` 7 times within our program. Let's try to do this in an elegant way using the techniques we've recently learned about.

```
def get_input():
    data = [0] * 7
    for i in range(7):
        data[i] = int(input())
    return data

arr = get_input()
print(arr)
```

Now we've set up some code to read in the input and show us what we've done. Let's confirm, using the terminal, that this does what we intend it to.

```
$ python soln.py < input.txt
```

```
[2, 2, 11, 4, 9, 7, 9]
```

Awesome – we can keep working on our solution. Let's work on the tail end of our program now. Eventually we're going to need to print our answer, so it'll be nice to have a function that handles this for us. Let's expand `soln.py`.

```python
def get_input():
    data = [0] * 7
    for i in range(7):
        data[i] = int(input())
    return data

def produce_output(x, y, z):
    print(x)
    print(y)
    print(z)

arr = get_input()
produce_output(1, 2, 5)
```

And we can check via the terminal that this indeed prints `1`, `2`, and `5` on separate lines. So far so good. Now comes the hard part – actually writing the solution.

```python
def get_input():
    data = [0] * 7
    for i in range(7):
        data[i] = int(input())
    return data

def produce_output(x, y, z):
    print(x)
    print(y)
    print(z)

def solution(data):
    # Now we need to be clever...

arr = get_input()
a, b, c = solution(arr)
produce_output(a, b, c)
```

Okay, now we need to do some thinking. One place to start is to look at the example input/output and try to pick up on a pattern. In the example, x and y turn out to be the two smallest numbers among the 7 input numbers. Maybe there's nothing there, but we can try to explain this behavior formally and precisely.

Let's think about the relationship between x and every other term among the seven. Since they're all positive, and x ≤ y, z, x will be less than x + z, x + y, y + z, and x + y + z. So we've shown that it will *always* be the smallest number among the bunch. Awesome, we're 1/3rd of the way there!

In fact, nearly identical reasoning shows that y will always be the second-smallest value among the seven, since y ≤ x + z, y ≤ y + x, and so on. Now we're almost done, we just need z. We might guess that z is the 3rd-smallest number, or even the biggest number, but the example we have shows that neither of those are true. One observation is that now we know x and y, so we know x + y. Furthermore, we know x + y + z, since it will be the largest number among the bunch (as x, y, z are positive). So we can find x + y + z and subtract x + y to arrive at z, and we're done!

Now we've done the hard thinking, and we just need to translate this idea into code. Let's focus on our function `solution()` for the moment, as we've written the rest of our file. Let's start off by finding the largest value in our input array, as we'll need it to compute z later.

```python
def solution(data):
    max = -1
    for i in range(7):
        if data[i] > max:
            max = data[i]
```

What we've written will kick things off by calculating the largest value in the array (make sure you understand why!). We can do something similar to find the smallest and second-smallest values in an array, but that's a bit of a hassle. Is there an easier way to do this (perhaps using something we learned last lecture...)?

Yes! We can sort our array, so that we know exactly where the largest, smallest, and second-smallest values are.

```python
def solution(data):
    data = sorted(data)
    x = data[0]
    y = data[1]
    biggest = data[len(data) - 1]
    z = biggest - (x + y)
    return x, y, z
```

Awesome, now let's finish up the rest of soln.py, using our previous functions for reading input and printing output. Our final answer will look like:

```python
def get_input():
    data = [0] * 7
    for i in range(7):
        data[i] = int(input())
    return data

def produce_output(x, y, z):
    print(x)
    print(y)
    print(z)

def solution(data):
    data = sorted(data)
    x = data[0]
    y = data[1]
    biggest = data[len(data) - 1]
    z = biggest - (x + y)
    return x, y, z

arr = get_input()
a, b, c = solution(arr)
produce_output(a, b, c)
```

§12 Monday, February 14

§12.1 HW3 Postmortem

People did much better on the quiz last week - nice. One remark about the last homework: many people wrote complicated code for the `improvement_needed()` function, relying on a whole bunch of casework. That might be correct (i.e., produce the right output) but there's a way to do it very succinctly, by observing that the required increases to SAT and ACT scores don't depend on one's class rank or GPA.

Check out the HW3 solution on Canvas to see what we mean. (Looking at the HW solutions in general might be helpful for learning about programming.) Furthermore, thinking about how to solve this problem elegantly will help sharpen the problem-solving skills that are central to computer science (e.g., organizing information, recognizing symmetries in a problem, etc.).

§12.2 Syntax

Recall that the **syntax** of a programming language is analogous to the the grammar of a natural language like English, while its **semantics** corresponds to the *meaning* associated to grammatically correct sentences. After all, we don't write grammatically correct sentences for the sake of writing them – the goal is to encode information.

Let's try to be a bit more formal about this 'grammar', or syntax, of Python. In English, we might define the syntax of a sentence as follows: a sentence consists of a subject followed by a verb following by a complement. We could write this definition as follows.

```
sentence :
        |subject verb complement
```

Being rigorous about the syntax of natural languages is a pretty tall order, though. It can only really be done approximately, and we leave this difficult task to linguists. Programming languages, meanwhile, have much stricter syntax. A program is simply a succession of statements, each on its own line. Now we need to define a statement – fortunately, there are only nine of them that you need to know!

```
statement :
        |assignment
        |expression
        |return_stmt
        |import_stmt
        |raise_stmt
        |function_def
        |if_stmt
        |for_stmt
        |while_stmt
```

Now we can (and need to!) go one level deeper. An assignment is simply a statement of the form `NAME` = expression. Now what's an expression? We need to go a couple layers deeper.... See the Jupyter notebook posted online for more detail here (not just for assignments and expressions, but for each of the other statements).

§12.3 Errors

We've been talking a bit about raising errors in our programs, and it's worth mentioning that errors fall into two camps: **static errors** and **dynamic errors**. A static error is one that Python will catch and report to you even before a program is executed. So syntax errors are static errors, for instance. Dynamic errors, meanwhile, occur only while the program is being executed, and may depend on the particular inputs that your program is being run on. For instance, `IndexError` s are dynamic errors.

Static errors are easy to find - just try to use your code once and you'll run into the error. Dynamic errors are a bit harder to find, and require that you test your code on a variety of inputs that, for instance, hit each of its branches. Testing isn't the most fun thing in the world, but it's very important, and you should be sure to check your homework solutions on at least a handful of different inputs before submitting.

§12.4 Semantics

Now that we've established some rules regarding the syntax of our programs, we can think about what those programs actually *mean*, i.e, what they're doing. Let's start with an assignment statement.

```
NAME = e
```

That statement has the following meaning in Python.

1. Evaluate the expression e, resulting in the value `v`.
2. Remember that the variable `NAME` has the value `v`.

That probably agrees with your intuition anyway, but the point is that there is an underlying translation between lines of code and their meaning, i.e., what the code actually does. Other, slightly more sophisticated tools in Python might have semantics that you find a bit surprising at first.

For instance, one common mistake that we see is that people confuse defining a function with calling a function. Defining a function simply informs Python that a function exists, and that it might be asked to use later. Calling a function has Python actually use the function, i.e., to get an answer for particular input(s)! Again, see the notebook online for detail on the semantics of many other statements.

§13 Wednesday, February 16

First things first: in order to be absolutely sure that everyone can complete the quizzes we've been having, we're going to create another one with much looser time limits. We'll release a small quiz tomorrow and you'll have several days to complete it and submit it in Canvas. As we've been saying for a while now, it's important for you to be comfortable with this process by the time the midterm comes around (in only two weeks!).

One more comment: you should really consider collaborating on your homework with other people. To be clear, you still need to write your own code and understand what's going on (and you can't look at your friend's code!), but it can be very useful to bounce ideas off another person and improve together.

§13.1 Problem: pairwise sum

We're going to start things off with another puzzle – here's the statement. You are given an integer S and an array of integers A[7]; now determine the number of pairs of entries in A whose sum is S.

The first, not-so-clever solution is what you might call a **brute force** solution, i.e., simply check all the pairs of elements in A and see if they sum to S. Before we even get there, let's kick things off by writing the part of our solution that will read input.

```python
def get_input():
    # Get S and N = len(A)
    S = int(input())
    N = int(input())

    # Grab entries of A one-by-one
    arr = [0] * N
    for i in range(N):
        arr[i] = int(input())
    return S, arr
```

[7] You receive these line-by-line from a `.txt` file, as in the last few examples and in the homework from now on.

```
def produce_output(result):
    print(result)
```

And as always, we'll quickly test this code to make sure it works (at least on a few examples). We also added a function for outputting our answer, even though it's pretty trivial in this case. So now we just need to do the heavy lifting and compute our answer from the inputs.

As we mentioned, one solution is via brute force – simply look at all possible pairs of numbers in A and see which of them sum to S. In particular, compare the first number in A with the other $N-1$ numbers, the second number in A with the other $N-2$ numbers (you've already checked the first and second numbers!), and so on. At a high level, we can see that we're going to be making

$$(N-1)+(N-2)+(N-3)+\cdots+1$$

many comparisons. This gives us a handle on how efficient our algorithm will be. Now how do we actually implement this?

Well, we're going to need to iterate over each of the entries in A. Then for each entry in A, we'll need to again iterate over all of the entries in A that lie to the right of it. This will look as follows.

```
def solution(S, data):
    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            print('i', i, 'j', j)
```

We're starting off by printing to make sure that our nested `for` loops are doing what we intend them to. We tested on an example and it looks like they indeed are, so we can keep moving forward. (Make sure that you understand the nested for loops we wrote above! The idea is to check each pair of distinct elements in `data` exactly once.)

```
def solution(S, data):
    count = 0
    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            if data[i] + data[j] == S:
                count += 1
    return count
```

Now we can tie everything together.

```
def get_input():
    S = int(input())
    N = int(input())
    arr = [0] * N
    for i in range(N):
        arr[i] = int(input())
    return S, arr

def produce_output(result):
    print(result)

def solution(S, data):
    count = 0
    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            if data[i] + data[j] == S:
                count += 1
    return count

S, arr = get_input()
result = solution(S, arr)
produce_output(result)
```

And boom – that's our solution!

§14 Friday, February 18

§14.1 Problem: pairwise sum (continued)

So, last time we implemented a 'brute force' solution for the pairwise sum problem. That's a perfectly fine solution, but it's not very efficient. This time around, we're going to think about making our solution faster by being a bit more clever. As we've mentioned a couple times, one of the primary tools we have in our kit is that of *sorting* our input. So, with the power of sorting in mind, spend a couple minutes thinking about how you might implement another solution for this problem.

Here's the idea: we start off with an index at the first entry in our (sorted) array and another at the last entry of the array. We add those values up – if that sum is greater than S, then we make it smaller by moving the rightmost index to the left by one (which can only make the sum smaller). If the sum is smaller than S, then we make it bigger by moving the leftmost index to the right by one (which can only make our new sum bigger). Make sure you understand why this argument makes use of our array being sorted, and why we won't miss any possible pairs that sum to S with this procedure.

Okay, let's write this up. Once again, we need an outer layer of code receiving input/printing output.

```
def get_input():
    S = int(input())
    N = int(input())
    arr = [0] * N
    for i in range(N):
```

```python
        arr[i] = int(input())
    return (S, arr)

def solution(S, data):
    # be clever now

S, data = get_input()
result = solution(S, data)
print(result)
```

Now let's focus on our solution.

```python
def solution(S, data):
    data = sorted(data)
    count = 0
    left = 0
    right = len(data) - 1

    # be clever
```

Okay, now we've sorted our data and set our indices `left` and `right` at the leftmost and rightmost entries in the sorted array. Now comes the heavy lifting, where we compute sums at each step and moving the indices left/right depending on what happens. Lots of things can happen at each step – maybe `left` goes up, maybe `right` goes down, maybe they both move – so it would be hard to write this with a `for` loop.

We're going to use a `while` loop instead, and we'll use `pass` as a stand-in for code that we haven't written yet.

```python
def solution(S, data):
    data = sorted(data)
    count = 0
    left = 0
    right = len(data) - 1

    while left < right:

        if data[left] + data[right] == S:
            pass
        elif data[left] + data[right] < S:
            pass
        else:
            pass
    return count
```

Now our code recognizes that there are three fundamental cases based on the value of `data[left] + data[right]` in relation to `S`. Let's do some more thinking and replace those `pass`'s with real code.

```
1   def solution(S, data):
2       data = sorted(data)
3       count = 0
4       left = 0
5       right = len(data) - 1
6
7       while left < right:
8
9           if data[left] + data[right] == S:
10              left += 1
11              right -= 1
12              count += 1
13          elif data[left] + data[right] < S:
14              left += 1
15          else:
16              right -= 1
17
18      return count
```

Okay, we have code that runs – nice. Now we tried it on a small example, and it agrees with our previous brute force solution, so we know it works, right! Wrong. At a bare minimum, you should test your code on a variety of examples, and even then you won't be sure that it works.

In this case, we try it on another example and see that our new `solution()` produces far smaller output than the brute force method. Okay, our answer is probably wrong.[8] Can you see what's going wrong? On an input like

```
S = 5
A = [2, 2, 2, 3, 3, 3]
```

our new solution will output 3, when the real answer is 9! The error lies in lines 9 through 12. Our code fails to count correctly when there are repeated values in the data. We need to update our code to keep track of these repeated values.

```
def solution(S, data):
    data = sorted(data)
    count = 0
    left = 0
    right = len(data) - 1

```

[8] We should have more faith in the brute force solution than our new solution, as the brute force solution is quite a bit simpler, both theoretically and in implementation/code.

```
    while left < right:

        if data[left] + data[right] == S:
            v_left = data[left]
            left_count = 0
            while data[left] == v_left:
                left += 1
                left_count += 1

            v_right = data[right]
            right_count = 0
            while data[right] == v_right:
                right_count += 1
                right -+ 1

            count += left_count * right_count

        elif data[left] + data[right] < S:
            left += 1
        else:
            right -= 1

    return count
```

And now we test on our example and we get the same answer as the brute force solution. We still don't know for sure that our new solution is correct, but at least we can be a bit more confident.

But this new solution took a lot more thinking and coding than the brute force solution – what was the point? Well, it turns out that our new solution is much, much more efficient than the brute force solution. That is, as inputs get larger, the new solution will run much more quickly than the brute force solution. The ideas behind making this notion of 'efficiency' formal – known as the *analysis of algorithms* – will be the subject of the next several lectures.

§15 Monday, February 21

§15.1 Analysis of algorithms

Last time, we created two solutions for the pairwise sum problem and mentioned briefly that one is more efficient than the other, meaning that it runs in less time. One of today's goals is to dig deeper into this idea, and to talk about the *analysis of algorithms*.

Let's take things from square one – how can we compare the runtimes of two different algorithms? Perhaps the most obvious thing to do is to simply race them against each other, i.e., to run them on equal input of increasing size and see how their runtimes evolve. That's a pretty reasonable idea, and it'll certainly tell us something about the algorithms, but we've learned from experience that there's a much better way of doing things.

The key observation is that any two algorithms we look at consist of the same fundamental building blocks, known as **elementary operations**. Examples include adding integers, setting an element of an array, testing whether something is `True` or `False`, etc. The idea is that these elementary operations are nearly the most basic things your computer can do – they can't be broken down very much into simpler operations. So if we can just count roughly how many basic operations each algorithm uses, we can get a good sense of its run time.

A key idea here is that we're going to be abstracting away lots of the low-level detail and minutiae – we won't keep track of the fact that some of these basic operations take a bit longer than others, and we won't even try to count the *exact* number of operations our algorithms take. Rough approximations will be enough.

Simply put, we want to drive home two points today:

1. We're measuring an algorithm's runtime behavior as a function of its input size.
2. We're counting the number of (some) elementary operations that the algorithm makes when computing the answer for an input.
    - Not *every* operation, just some! As a rule of thumb, you should count whichever operation is going to be executed the most in the algorithm.

Let's get a little more concrete (though things will still be abstract). Let's say we have the following function.

```python
def f(x):
    g(x)
    h(x)
    k(x)
```

What is the runtime behavior of `f`? Well, Python will evaluate its body line-by-line when `f` is called, so its runtime is just the sum of the runtimes of `g`, `h` and `k`! Please make sure you understand this before moving forward. Now we'll be even more concrete. What about this function?

```python
def f0():
    counter = 0
    counter += 1
    counter += 1
    counter += 1
    counter += 1
    counter += 1
    print(counter)
```

This is a bit of a strange case, because `f0()` doesn't even take input. So what `f0` prints should just be a fixed number, which we can check will be 5. In fact, `counter` is incremented 5 times, regardless of the size of the input to `f0()` (since `f0()` doesn't even take input!). We would say that `f0()` is a *constant time* algorithm.

Another example:

```python
def f1(N):
    counter = 0
    for i in range(N):
        counter += 1
    print(counter)
```

In this case, by understanding the semantics of `for` loops, we can see that `counter` is going to be incremented `N` many times. We would say that `f1` is a *linear time* algorithm.

```
def f2(N):
    counter = 0
    for i in range(N):
        counter += 1
        counter += 1
    print(counter)
```

Now, the number of basic operations taken by `f2` in input N is $T(N) = 2 \cdot N$. (We're simply using T to count the number of these operations for a given algorithm). Once again, we would say that the number of operations performed by `f2` is linear in its input size.

```
def f3(N):
    counter = 0
    for i in range(N):
        counter += 1
    for j in range(N):
        counter += 1
    print(counter)
```

Once more, we'll have that $T(N) = N + N = 2 \cdot N$, with one N term coming from each of those loops. The next example is an important one.

```
1  def f4(N):
2      counter = 0
3      for i in range(N):
4          for j in range(N):
5              counter += 1
6      print(counter)
```

Well, the loop beginning on line 4 has a runtime of N, and it occurs N many times, so our overall operation count is

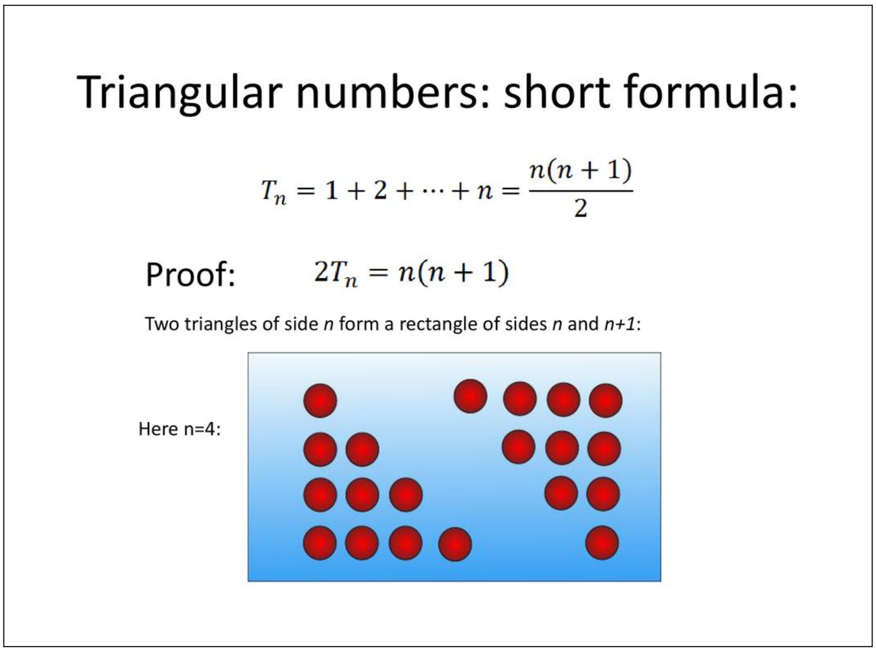
$$T(N) = \overbrace{N + \cdots + N}^{N \text{ times}} = N^2.$$

```
def f5(N):
    counter = 0
    for i in range(N):
        for j in range(i+1, N):
            counter += 1
    print(counter)
```

Similarly, the runtime here is

$$T(N) = (N-1) + (N-2) + \cdots + 1 = \frac{N\cdot(N-1)}{2}.$$

To see why that formula holds, here's a proof in a picture.



Triangular numbers: short formula:

$$T_n = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Proof: $2T_n = n(n+1)$

Two triangles of side *n* form a rectangle of sides *n* and *n+1*:

Here n=4:

Figure 1: Formula for triangular numbers, from here.

§16 Wedneday, February 23

§16.1 Analysis of algorithms, continued

Last time we started talking about the analysis of algorithms, and we used the function $T(N)$ to denote (roughly) how many operations our algorithm performs on input N. We chose to count the number of times that the `counter` variable is incremented in the functions we examined last time. Recall the big idea - we want to count the operation that is performed most frequently by our algorithm, in order to understand its runtime behavior.

Here's another example, where there's more than one input.

```
def f(N, M):
    counter = 0
    for i in range(N):
        for j in range(M):
            counter += 1
    print(counter)
```

In this case $T(N,M) = N \cdot M$. Make sure that you understand why - the inner loop contributes a term of M to the counter, while the outer loop causes the inner loop to be repeated N many times. So,

$$T(N,M) = \overbrace{M + \cdots + M}^{N \text{ times}} = N \cdot M.$$

Now let's think about the runtime behavior of our brute force solution to the previous pairwise sum argument. Recall that the core of our solution was as follows:

```
1 def solution(S, data):
2     count = 0
3     for i in range(len(data)):
4         for j in range(i + 1, len(data)):
5             if data[i] + data[j] == S
6                 counter += 1
7     return counter
```

This time around, we're actually going to count the number of times that line 5 is executed by our program, since that will be the most common operation.From our previous discussion of 'triangular numbers', we can see that

$$T(\text{S, data}) = 1 + 2 + \cdots + (\text{len(data)} - 1) = \frac{\text{len(data)} \cdot (\text{len(data)} - 1)}{2}.$$

Note that this is totally agnostic of what the actual entries of `data` are; the runtime behavior depends only on its size, not the actual contents.

Remark 16.1. Technically, line 5 has a couple of operations bundled up: finding `data[i]`, finding `data[j]`, adding them, and checking whether they equal S. But for our purposes, there's not much of a difference between (for instance) $T(S) = S$ and $T(S) = 4S$, since they grow in essentially the same way as S increases. What we really want is to distinguish $T(S) = S$ from something like $T(S) = S^2$. Those functions grow very differently as S increases.

Another example.

```
def f(N):
    counter = 0
    if N % 2 == 0:
        for i in range(N):
            for j in range(N):
                counter += 1
    else:
        for i in range(N):
            counter += 1
    print(counter)
```

In this case, $T(N)$ (the number of times that `counter` is incremented) actually depends considerably on whether N is even or odd. We have

$$T(N) = \begin{cases} N^2 & N \text{ is even,} \\ N & N \text{ is odd.} \end{cases}$$

(Make sure you can see why.) How do we handle this kind of branching? The short answer is that computer science often defaults to **worst-case analysis**. That is, it usually studies how bad things can possibly get, as opposed to the best-case or the average.[9] So, using this idea, we would say that the previous algorithm is $T(N) = N^2$ in the worst case.

Let's look at another example, keeping this worst-case philosophy in mind.

```
def f(array):
    counter = 0
    for i in range(len(array)):
        if array[i] % 2 == 0:
            counter += 1
```

We're actually going to count two things this time; let $A(N)$ be the number of array accesses for an input array of size N and $B(N)$ be the number of counter increments for an input array of size N. We can see instantly that $A(N) = N$, guaranteed. Describing $B(N)$ requires some worst-case analysis, since it depends on the number of even numbers in our array. In the worst case, though, all its entries are even, meaning $B(N) = N$ (again, *in the worst case*!).

Another example, similar to the previous one but different.

```
def f(array):
    counter = 0
    for i in range(len(array)):
        if i % 2 == 0:
            counter += 1
```

In this case, the behavior of the algorithm doesn't depend on the input array at all! We just have $T(N) = N/2$, in every case (again, we're counting the number of times that `counter` is incremented). Another example, that looks suspiciously close to some of the code we wrote in our clever solution to the pairwise sum problem...

```
def f(array):
    left = 0
```

[9] Though there is certainly such a thing as average-case analysis of algorithms!

```
    right = len(array) - 1
    v_left = array[left]
    v_right = array[right]
    while left < right:
        if v_left <= v_right:
            left += 1
            v_left = array[left]
        else:
            right -= 1
            v_right = array[right]
```

The idea is that `left` and `right` begin at opposite ends of the array and creep towards each other, step by step. We can't really say exactly where they'll meet in the array, but the key idea is that the sum of steps taken by `left` (i.e., the number of times it is incremented) and `right` (i.e., the number of times it is decremented) is constant. It has to add up to the length of the array! So $T(N) = N$ here, where we're counting the number of times that `left` or `right` are changed.

Another example, getting quite close to the code we wrote for the pairwise sum problem.

```
def f(array):
    left = 0
    right = len(array) - 1
    v_left = array[left]
    v_right = array[right]
    while left < right:
        if array[left] == array[right]:
            left += 1
            right -= 1
            v_left = array[left]
            v_right = array[right]
        elif array[left] > array[right]:
            left += 1
            v_left = array[left]
        else:
            right -= 1
            v_right = array[right]
```

Using similar reasoning, we can again see that $T(N) = N$, as `left` and `right` will need to travel a total distance of N in order to meet and thus conclude the `while` loop.

§17 Monday, February 28

§17.1 Analysis of algorithms, continued

We'll be continuing our discussion on the analysis of algorithms today. This will be the final lecture exclusively dedicated to the subject, though we'll certainly be encountering it throughout the remainder of the semester. Recall the high-level goal: we want to understand the runtime behavior of our algorithms. Simply put, how long do they take to run?

We're abstracting away from lots of the low-level minutiae of our programs – including details of the hardware on which they're running – and simply counting the rough number of 'basic operations' taken by our algorithm. As a rule of thumb, we want to count the basic operation that is performed most frequently by our algorithm (e.g., incrementing a counter, updating the element of an array, etc.). The second layer of abstraction is that we don't pay attention to the exact details of the algorithm's input: we only care about its size. In the case in which the size of an input doesn't uniquely determine its runtime under our algorithm, we default to the worst-case scenario. Simply put, what is the *longest* that our program will run on an input of size N?

Another layer of abstraction that we'll introduce today is that we don't care about small, constant terms. For instance, runtimes of $T(N) = N$ and $T(N) = N + 3$ are essentially the same for our purposes. As N gets large, these small differences will quickly wash out. Perhaps even more surprisingly, we'll say that $T(N) = N^2 + N$ and $T(N) = N^2$ are essentially the same, despite the fact that N will tend to infinity as input grows. The reason why is that both of those runtimes will be dominated by the N^2 term as N grows. For instance, when $N = 100$ (which is not that big), $N^2/(N^2 + N) > 99\%$. That N term just really doesn't matter!

The point is that we really only care about the **asymptotics** of these functions, i.e., their behavior as N tends to infinity.

Example 17.1

If $T(N)$ is some polynomial in N, i.e., $T(N) = a_k N^k + a_{k-1}N^{k-1} + \cdots + a_0$, then we really only care about the largest-degree term N^k. For instance, the runtimes $T(N) = N^3$ and $T(N) = N^3 + 10000 \cdot N^2$ are essentially the same. They're each dominated by N^3.

So we want some tool for compressing all of this information and telling us *only* what we need to know about $T(N)$. For instance, the tool would tell us that the only thing that really matters about $T(N) = 2N^3 + 10000 \cdot N^2 + 50$ is that it has an N^3 term. The perfect tool for this is **big O notation**. It will allow us to – in a meaningful way – ignore these smaller-order terms. At a high level, $T(N) \in O(g(N))$ means that $T(N)$ is less than $g(N)$ once the input gets large enough.[10]

Computer scientists think of big O roughly as a ≤ sign, i.e., $T(N) \in O(g(N))$ can be interpreted as $T(N)\text{“}\le\text{”}g(N)$. Furthermore, computer scientists often play fast and loose with the equals sign and write $T(N) = O(g(N))$, though strictly speaking they mean that $T(N)$ is a member of the set $O(g(N))$. (In particular, you would never want to swap the sides of that equality sign and write $O(g(N)) = T(N)$.) We'll use both of those notations on occasion, and we'll also write "$T(N)$ is in $O(g(N))$" to mean $T(N) \in O(g(N))$ or $T(N) = O(g(N))$.

[10] Strictly speaking, less than some *multiple* of $g(N)$, but we won't worry too much about the details here.

Example 17.2

Say $T(N) = a_k N^k + a_{k-1} N^{k-1} + \cdots + a_0$. Then $T(N) \in O(N^k)$, agreeing with our intuition from the previous example! So, for instance, $T(N) = 0.125N^3 + 45N^2 + 11$ is in $O(N^3)$.

Example 17.3

All constants are in $O(1)$. For instance, $2034 \in O(1)$. Intuitively, they're just numbers that don't depend on N and get washed out as N grows to infinity. Another way of thinking about it is that $2034 = 2034 \cdot N^0$, so $2034 \in O(N^0)$ by the previous example, but N^0 is just 1.

Here's an exercise – what does big O analysis say about the following function?

```python
def f():
    counter = 0
    counter += 1
    counter += 1
    counter += 1
    counter += 1
    counter += 1
    print(counter)
```

$T(N) = 5$, so this algorithm is in $O(1)$, by the previous example. Another exercise.

```python
def f(N):
    counter = 0
    for i in range(N):
        for j in range(i+1, N):
            counter += 1
    print(counter)
```

Previously, we've seen that $T(N) = (N-1) + (N-2) + \cdots + 1 = \frac{N \cdot (N-1)}{2}$. So `f` is in $O(N^2)$. What about this next one?

```python
def f(N):
    counter = 0
    for i in range(0, N, 4):
        for j in range(i+2, N-4, 3):
            counter += 1
    print(counter)
```

Once more, our program has two nested `for` loops of size proportional to N, so the algorithm is in $O(N^2)$. We really don't need to worry about the fact that one loop only goes to $N-4$, or that they have step sizes bigger than 1. This will only change $T(N)$ up to some constant coefficients, so it doesn't matter for the sake of big O.

One important note: if an algorithm is in $O(N^2)$, then it's also in $O(N^3)$ and $O(N^4)$, and so on. Intuitively, if $T(N)$"$\leq$"N^2 then certainly $T(N)$"$\leq$"N^3 and $T(N)$"$\leq$"N^4. So there are many choices of the big O description we could have for $T(N)$, but we'd like to give the 'tighest' bound for it, i.e., to write $T(N) = O(g(N))$ where $g(N)$ is as small as possible.

For instance, if $T(N) = 0.25N^2 + 3N - 20$, then $T(N) \in O(N^2)$ is tight, since $T(N)$ indeed has an N^2 term. But $T(N) \in O(N^3)$ is not tight, since you can say something even more informative about $T(N)$. Of course, you try to give tight bounds whenever possible, though sometimes loose bounds are the best you can do.

Example 17.4

The sorting function in Python is in $O(N \log N)$. So, if you were to plot the runtime of `sorted(arr)` as the length of `arr` grows, you would see a curve that looks like $N \log N$.

Using the result from the previous example, note that the following algorithm is in $O(N^2)$, as $T(N) = N^2 + N \log N$ is in $O(N^2)$.

```
def f(arr):
    arr = sorted(arr)
    count = 0
    for i in range(len(arr)):
        for j in range(len(arr)):
            counter += 1
    print(counter)
```

This big O notation will show up on the midterm this week, so try to understand these examples! One last note on nomenclature. If an algorithm is in $O(1)$, then it is said to be in constant time, while algorithms in $O(\log N)$ are said to have logarithmic runtime. More generally,

- $O(1)$: constant,
- $O(\log N)$: logarithmic,
- $O(N)$: linear,
- $O(N \log N)$: log-linear,
- $O(N^2)$: quadratic,
- $O(N^3)$: cubic,
- $O(2^N)$: exponential.

§18 Monday, March 14

§18.1 Midterm postmortem

Midterm grades were posted this morning, and the scores were pretty good overall. The second part of the midterm was quite challenging (by design!), but a handful of people were able to find the efficient solution that we had in mind. If you were able to find it, that's fantastic, but if you weren't able to find it then it's not too big of a deal. There's lots of time left to keep learning about these concepts, and in fact most people weren't able to find the efficient solution in only 45 minutes.

§18.2 Problem: 3Sum

Now let's think about the 3sum problem from Part 2 of the midterm, i.e., given an array, find the number of triples of entries in the array that sum to 0.

Here's the not-so-clever solution to the problem.

```
def sum3(A):
    N = len(A)
    count = 0
    for i in range(N):
        for j in range(i+1, N):
            for k in range(j+1, N):
                if A[i] + A[j] + A[k] == 0:
                    count += 1
    return count
```

That would earn you 7/10 points on the midterm, as it's in $O(N^3)$ but not in $O(N^2)$. The more clever solution is as follows: note that `A[i] + A[j] + A[k] == 0` is exactly the same thing as `A[i] + A[j] == -A[k]`. So the problem really just amounts to repeatedly applying the 2Sum solution that we already figured out in class.

And copying/pasting code from class was allowed on the midterm, so you indeed could have used the solution we made in class (with some minor tweaking).

```
def sum2(S,data,start):
    count = 0
    left = start
    right = len(data) - 1
    while left < right:
        if data[left] + data[right] == S:
            count += 1
            left += 1
            right -= 1
        elif data[left] + data[right] < S:
            left += 1
        else:
            right -= 1
    return count

def sum3(arr):
    arr = sorted(arr)
    count = 0
    for i in range(len(arr)):
        count += sum2(-arr[i],arr,i+1)
    return count
```

At a high level, we tackled this 3Sum problem by using an existing solution for a related problem, the 2Sum problem (also called the pairwise sum problem). This kind of idea is known as a **reduction** in computer science. The idea is that a problem A 'reduces' to problem B if you can use a solution for B to get a solution to A. One takeaway is that your mental reservoir of solutions will grow as you solve more problems, and this will making solving additional problems easier and easier!

§18.3 Leetcode

Some of you have been asking about how you can get more practice solving algorithmic problems: that's great! One resource to know is https://leetcode.com. It has tons of problems of varying difficulties – it's probably best to focus on easy and medium level problems for now – and a great interface for writing your code, testing it on examples of your choosing, etc.

It even has weekly contests, problems organized by topic, entire libraries of problems for mastering certain techniques, etc. You might find that working on these leetcode problems is actually really fun (especially once you solve them!), and it'll certainly make you a better programmer. There's even a chance that the final project will consist of having you solve 20 or 30 of these problems!

§18.4 Search problems

Now let's think about search problems. The idea is that you have a container of information, say an array, and you want to know whether a value x lives inside this container. For an array of length N, you may need to check every single entry to know whether x is an entry of the array, so there's a linear cost to this search in general (i.e., $O(N)$). Formally, the algorithm is something like:

```
def is_member(x, A):
    for i in range(len(A)):
        if A[i] == x:
            return True
    return False
```

But what if the array A is sorted? Then this is a bit like looking for someone's name in a directory. In real life, you wouldn't look for John Smith in a directory by checking page 1, then page 2, then page 3, and so on. You would probably open the directory roughly in the middle and check to see if the names you're seeing are before or after 'John Smith'. If they were names that come before 'John Smith', you'd forget about the first half of the directory and restart your search on the second half. (And likewise if you were to see names that come after 'John Smith'.) In this way, you've cut your problem in half with only one step of computation!

So, in the worst case, how many times do you need to cut your problem in half until you're done, i.e., until you're left with a single page among the N pages? That would just be $\log_2(N)$. This is *way* faster than the linear scan we talked about a moment ago!

There's a name for this kind of procedure: it's known as **binary search**. It's an extremely powerful technique for searching on sorted data, and it often shows up even in settings that seemingly have nothing to do with searching on sorted data. (Spoiler: you'll encounter binary search again on this week's homework.)

§18.5 Sorting

As we've seen time and again, sorting your data is often a crucial move when solving a problem. We've told you that the fastest solution for sorting is in $O(N \log N)$ – that is, there is a log-linear solution – but we still haven't told you how to actually sort an array. One sorting algorithm we'll discuss now is **insertion sort**. It's not the fastest, but it'll still help you see how to sort an array, and it'll introduce you to the idea of recursion, which we're going to start seeing a lot of.

Here's the idea behind insertion sort. Let's assume we have an array of size N and the first k many entries are sorted. In order to improve upon this and make it so that the first $k+1$ entries are sorted, we can look at the $(k+1)$th entry (i.e., `A[k]`) and place it in the right spot among the first k sorted entries. How we can find where the $(k+1)$th entry goes among the first k? Using binary search!

An important idea here is that we are *assuming* that we have a way of sorting the first k entries, and we're then using that to build an algorithm for sorting the first $k+1$ entries. (Crucially, the 'base case', in which we need to sort just 1 entry, is trivial – an array of length 1 is automatically sorted.) This idea, of using a solution as part of itself, is known as **recursion**, and we'll become quite familiar with it over the rest of the course.

§19 Wednesday, March 16

§19.1 Insertion sort

The name of the game is to sort an array of integers. For instance, we want an algorithm that takes in the array

```
[8, 1, 7, 2, 5]
```

and spits out the array

```
[1, 2, 5, 7, 8]
```

(without using a built-in function like `sorted()`). As we were mentioning last time, you've probably sorted a handful of cards in the past without having anyone explicitly teach you how to do it.

So, somehow you already have a solution for this within yourself, but the tough part now is to actually convert it into an effective piece of Python code. As part of this process, we're going to talk about the **divide and conquer** tactic for solving algorithm problems. This is a *really* important idea in computer science, and you'll be seeing a lot of it if you end up becoming a computer science major.

The big picture of divide and conquer looks like this:

1. Begin with a problem P.

2. Divide P into two strictly 'smaller' problems P' and P''.

3. Assume, by magic, that you have solutions S' and S'' for the problems P' and P''.

4. Use the solutions S' and S'' to construct a solution S for the original problem P.

Once again, this is a powerful idea in computer science that often gives rise to very efficient (i.e., fast) solutions, and hopefully you can see why it's called divide and conquer. Now let's return to the sorting problem, and to the insertion sort idea that we were talking about last time. We can describe the algorithm using the same divide & conquer format as above. We'll use the example from earlier to keep things a bit concrete.

1. The problem P is to sort `[8, 1, 7, 2, 5]`.

2. Let's divide P into P', the problem of sorting `[8, 1, 7, 2]` and P'', the problem of sorting `[5]`.

3. Assume, by magic, that we can solve P' and P''.
   - We don't even need magic to sort `[5]` – it's already sorted!

4. Use sorted copies of `[8, 1, 7, 2]` and `[5]` to build the sorted copy of `[8, 1, 7, 2, 5]`.
   - This is easy! On the one hand we have `[1, 2, 7, 8]` and on the other we have `[5]`. Since that first array is already sorted, we can easily find out where `5` should be placed inside of the array. Once we put `5` in the right spot, we'll have a sorted copy of our entire original array!

So the heavy lifting now is to write the function that puts `5` in the right place in `[1, 2, 7, 8]`. Formally, let's say we have an array `A` where the first entry of `A` is some arbitrarily value and the rest of the entries in `A` are sorted. Now let's write a function that sorts `A`.

```python
ex1 = [8, 1, 2, 3, 4, 5, 6, 7]
ex1 = [4, 1, 2, 3, 5, 6, 7, 8]

def insert(A):
    N = len(A)
    v = A[0]
    i = 1
    while i < N and A[i] < v:
        A[i-1] = A[i]
        A[i] = v
        i += 1
    return A

print(insert(ex1))
print(insert(ex2))
```

Make sure to understand what our function `insert()` is actually doing, and to convince yourself that it's right. We're not going to totally follow through here and complete the implementation of insertion sort (that will be your job on the next homework!), but hopefully you can see how this is one piece of our algorithm. Once again, the idea is to sort the first $k = 1$ entries of our array A (by doing nothing!), and then, as k increases, to sort the first k entries of A by simply placing the kth entry of A in the right spot among the first $(k - 1)$ already sorted entries.

Now, even though we haven't coded it up yet, let's think about the efficiency of this algorithm. After all, analyzing an algorithm's performance with big-O notation is extremely important and usually doesn't even rely on looking at actual code! In this case, we can see that our algorithm performs $O(N)$ insertions, each of which is an $O(N)$ operation, so insertion sort is in $O(N^2)$.

We told you a while back, though, that there is an $O(N \log N)$ implementation of sorting that Python uses in the `sorted()` function. How does that faster algorithm work? It's actually very similar to insertion sort! Instead of starting things off by splitting the sorting problem into one problem of size 1 and another of size $(N - 1)$, you instead split into two problems of size roughly $N/2$. If we had a linear-time way of merging our sorted halves of the array, then we would only be performing an $O(N)$ operation (i.e., merging) $O(\log N)$ many times (i.e., how many times you need to divide N by 2 in order to reach a 'base case' of size of 1). This kind of linear-time merging algorithm indeed exists, and this idea gives rise to the $O(N \log N)$ **merge sort** algorithm.

§20 Friday, March 18

§20.1 Merge sort

We're going to keep talking about **divide and conquer** algorithms today. Recall the big picture:

1. Begin with a problem P.
2. Divide P into two strictly 'smaller' problems P' and P''.
3. Assume, by magic, that you have solutions S' and S'' for the problems P' and P''.
4. Use the solutions S' and S'' to construct a solution S for the original problem P.

Step 2 is what we'd call the 'Divide' step and step 4 is the 'Conquer' step. In the case of insertion sort, we divided the problem P (sort an array A) into the problems P' (sort $A[0]$) and P'' (sort $A[1 : \mathrm{len}(A)]$). The conquering step amounted to having our insertion function that places $A[0]$ in the right spot among the sorted copy of $A[1 : \mathrm{len}(A)]$. That's why it's called insertion sort!

Last time we also talked about the complexity of insertion sort and concluded that it's in $O(N^2)$. Simply put, that was because A) the problem P is divided N many times until it reaches the base case where P'' has size 1, and B) each conquering step is in $O(N)$ in the worst case. So our overall algorithm takes N many $O(N)$ operations, meaning it's in $O(N^2)$.

We also started speaking briefly about an algorithm that is similar to insertion sort but much faster: **merge sort**. The idea is that in step 2 of the divide and conquer blueprint, we divide P into P' and P'' by splitting it down the middle (i.e., into the problem of sorting the first half of the array A and of sorting the second half), instead of splitting it into problems of size 1 and $N - 1$ (i.e., to sort $A[0]$ and $A[1 : \mathrm{len}(A)]$).

If we do this, then our conquering step will involve taking two sorted arrays and 'merging' them into a single sorted array, rather than just inserting one number into a sorted array (hence the names insertion sort and merge sort!). So how do we go about merging?

```python
def merge(A, B):
    N = len(A)
    M = len(B)
    C = [0] * (N + M)

    i = 0
    j = 0
    k = 0

    # Working with both A and B
    while i < N and j < M:
        if A[i] < B[j]:
            C[k] = A[i]
            i += 1
        else:
            C[k] = B[j]
            j += 1
        k += 1

    # A or B is empty
    if i == N:
        # A is empty
        while j < M:
            C[k] = B[j]
            j += 1
            k += 1
    else:
        # B is empty
        while i < N:
            C[k] = A[i]
            i += 1
            k += 1

    return C
```

Make sure you follow what's going on here: we're peeling off the smallest element in `A` or `B` (that we haven't yet seen), placing it in the sorted array that we're building, and continuing to move rightward in `A` and `B`. How can we see that the code has linear complexity, i.e. $O(N + M)$? Note that k can never exceed $N + M$, and that it is incremented after every chunk of computation.

Now that we've convinced ourselves that `merge()` is linear, let's think about the complexity of the merge sort algorithm overall. There are two questions here:

1. How many times will we need to divide our problem?
2. What is the cost of conquering at each step?

Since we're dividing the problem in half each time, the answer to (1.) is $\log N$. Simply put, you can only divide N by 2 a quantity of $\log N$ many times before you hit 1, the base case. Furthermore, we just saw that conquering (via `merge()`) is in $O(N)$, so the complexity of merge sort is $O(N \log N)$.

There's a point we want to drive home here: it may not be easy to see what the real difference is between $O(N \log N)$ and $O(N^2)$ off the top of your head, but the difference is *huge*. On input of size one million, it would take your laptop about 2.8 hours to run insertion sort and just 1 second to run merge sort. On input of size one billion (very reasonable for many practical settings), it would take your laptop 317 years to run insertion sort and just 18 minutes to run merge sort. Even a super computer worth millions of dollars would take a full week to run insertion sort on input of size one billion.

Bottom line: laptops with clever algorithms can demolish super computers with not-so-clever algorithms. Algorithmic thinking matters a lot!

§21 Monday, March 21

§21.1 Data structures and data types

Today, we'll be discussing another set of powerful tools for solving algorithmic problems: data structures and data types. As a brief primer, we'll be discussing all of the following things:

- Stacks,
- Queues,
- Lists,
- Sets,
- Dictionaries.

Being comfortable with such data types can often help you make your solutions considerably more efficient than they otherwise would be, and even to help you discover solutions to problems that are otherwise very unapproachable. To motivate things, let's think about the following problem.

Example 21.1

Given a string `s` containing just the characters `(, ), {, }, [, ]`, determine whether the input string is valid. An input string is valid if:

1. Open brackets are closed by the same kind of bracket, and
2. Open brackets are closed in the correct order.

For instance, `'()'`, `'(){}'`, and `'({})[]'` are valid, but `')('` and `'(]'` are not.

Think seriously about how to solve this problem; somehow it seems simple enough, but it's really not easy to solve using only the techniques that we currently know. It turns out, though, that it's generally considered to be quite an easy problem, once you have the right tool.

The tool we really want here is that of a **stack**. Abstractly, a stack is a data type where you can add values to the stack and remove the most recently added element. That's it - it's like a magical backpack where you're allowed to do two things: 1) place items in the backpack, and 2) ask the backpack to return the last thing that you gave it.[11]

Now, the notion that we've just described for the stack (i.e., as a collection of data with a few key operations) is known as a **data type**. It is simply an abstract description of an interface for handling data, as in the magical backpack that you can place items in and request for the most recently placed item. Notably, it does *not* include any implementation details of how such a thing could be done by a computer.

An actual implementation of such a data type is known as a **data structure**. Once again, the data structure includes the details of how the data type is actually implemented under the hood. Now let's use this stack data type to solve our problem.

```
def match(oc, cc):
    # Check if opening character (oc) and closing character (cc) match
    if cc = ')':
        return oc = '('
    elif cc = '}':
        return oc == '{'
    else
        return oc == '['

def solution(s):
    N = len(s)
    A = [''] * N
    top = 0

    for i in range(N):
        c = s[i]
        if c == '(' or c == '{' or c == '[':
            A[top] = c
            top += 1
        else:
            if top == 0:
                return False
            top -= 1
```

[11] Strictly speaking, we also need to be able to create the stack and to check if it's empty, but that's the easy part.

```
            v_top = A[top]
            if not match(v_top, c):
                return False

    return top == 0
```

This code should work, but it's pretty messy. The logic of our algorithm, which is very simple, is mixed up with the logic of implementing the stack. We'd really like to separate these things: in one place you'd have the data structure's details, and in another place you'd use the data type elegantly. So let's actually do this.

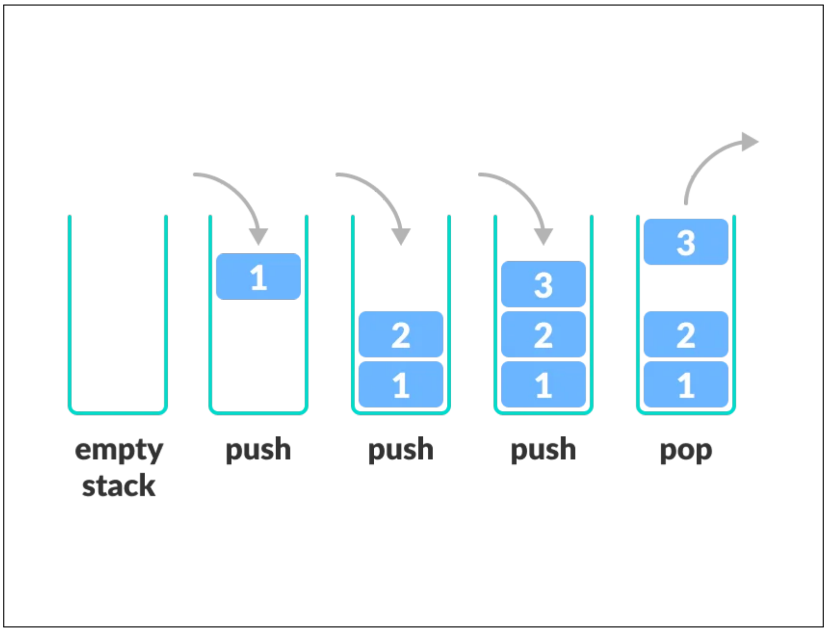


Figure 2: Visualization of a stack, from here.

We'll need functions to create a stack, push a value onto a stack, pop the most recent value from a stack, and check whether a stack is empty. This will make our code much cleaner, and it will let us use the stack as an abstract data type (i.e., without running into its implementation details in the middle of our algorithm).

```
def match(oc, cc):
    # Check if opening character (oc) and closing character (cc) match
    if cc = ')':
        return oc = '('
    elif cc = '}':
        return oc == '{'
    else
        return oc == '['

def create(size):
    A = [''] * size
    top = 0
    return (top, A)

def push(v, stack):
    (top, A) = stack
    A[top] = v
    top += 1
    return (top, A)

def pop(stack):
    (top, A) = stack
    top -= 1
    v = A[top]
    return (top, A)

def is_empty(stack):
    (top, A) = stack
    return top == 0

def solution(s):
    N = len(s)
    stack = create(N)

    for i in range(N):
        c = s[i]
        if c == '(' or c == '{' or c == '[':
            stack = push(c, stack)
        else:
            if is_empty(stack):
                return False
            (v_top, stack) = pop(stack)
            if not match(v_top, c):
                return False

    return is_empty(stack)
```

Our new solution is much cleaner – we've separated the data type from the data structure. We can now see that `solution()` is just using a stack in a fairly straightforward way. The code is now easier to read, write, and reason about. It's much less likely that we'll forget to increment a `top` counter somewhere, and we can even change our implementation of the stack data type (to something simpler or faster) without changing `solution()` at all!

§22 Wednesday, March 23

§22.1 The stack data type

Last time we talked about using the stack data type to solve the problem of parenthesizations being valid. As we implemented it, the stack data type came with four operations: creation of the stack, determining whether the stack is empty, pushing a value onto the stack, and popping a value from the stack.

The exact operations required by a stack can vary slightly across textbooks and websites (for instance, one source might require that we have a way to know how many elements are currently in the stack), but the key operations are really push and pop. It's a bit like a stack of trays: you can place a tray ontop of the stack, and you can pop off the most recently placed tray (whereas it'd be hard to grab a tray from the middle). Those two operations are really the essence of the stack data type.

We also talked about data structures last time, and implemented the stack data type using a data structure consisting of an array and index that kept track of the first unoccupied entry in our array. Let's remind ourselves of the code that we wrote.

```
def create(size):
    A = [''] * size
    top = 0
    return (top, A)

def push(v, stack):
    (top, A) = stack
    A[top] = v
    top += 1
    return (top, A)

def pop(stack):
    (top, A) = stack
    top -= 1
    v = A[top]
    return (top, A)

def is_empty(stack):
    (top, A) = stack
    return top == 0
```

Recall that we had to be deliberate about separating the logic of our data structure from the logic of our algorithm for solving the parenthesization problem. In addition to making our code easier to read and reason about, it also allows us to compartmentalize our data structure from our algorithm, in the sense that we can easily make our data structure cleaner or more efficient without perturbing our algorithmic solution.

So that was just one possible data structure for implementing the stack data type: using an array and an index. There's another way to implement the stack data type, using tuples.[12] How would we represent an empty stack as a tuple? Well, we can use the empty tuple `()`. And if we already have some stack S, we can represent v being pushed onto s by (v, S). This is a recursive idea, one of the key themes of the course!

Let's be a bit more concrete and see this in action. We can start with an empty stack `s = ()`. Now we can push 3 onto `s` and end up with `s = (3, ())`. We can then push 11 and get `s = (11, (3, ()))`. To pop from a given (non-empty) stack `s`, we can grab the value `s[0]` and replace `s` with `s[1]`.

It can be a bit mind-boggling, but try to understand why this idea works before seeing how we implement things in Python. We can now write:

```python
def create(size):
    return ()

def push(v, stack):
    return (v, stack)

def pop(stack):
    return stack

def is_empty(stack):
    return stack == ()
```

This implementation is much simpler than what we saw last time! The `pop()` function is maybe the most difficult to understand – the idea is that we want `pop(s)` to return a tuple where the first entry is the topmost element of `s` and the second entry is the rest of `s`. But `s` itself already takes this form! This data structure we've just implemented is really a **linked list**, in which values are connected to each other by pointers.[13]

Now, in order to make sure that our implementation of the stack is actually useful, we need to analyze its complexity. For instance, our implementation wouldn't be too useful if it turns out that `pop()` is in $O(N^{10})$ for instance (and not in $O(N^9)$).

Remark 22.1. When we think about the efficiency of the stack operations, we'll want to think about their efficiency in terms of the size of the underlying stack data structure, not necessarily the argument to the operation. In this case, all our functions take `stack` as an argument – because they need to! – so it's just the usual runtime analysis, but this remark will be more important when we get to object-oriented programming on Friday. All in due course.

So, what are the runtimes for our new implementation of the stack? Well, `create()` is certainly in $O(1)$, as it doesn't even use its argument, and `push()` simply creates a set with two entries so it is in $O(1)$. Likewise, `pop()` simply returns its argument and is in $O(1)$, and `is_empty()` tests for equality with `()` which is $O(1)$. This is is a great implementation – everything is constant time!

In fact, this beats our previous implementation, for which `create()` is in $O(\text{size})$. Furthermore, our two implementations can actually differ in their behavior! Our recent

[12] In fact, there are always many possible data structures for implementing a given type!

[13] This isn't too important for now, so don't worry if it doesn't quite make sense yet.

implementation has `push()` defined recursively, so you can place as many values on it as you'd like. Is that true for our older implementation? No – once we push more than `size` many elements onto our stack, we'll get an `IndexError` because we've filled our underlying array.

How can we fix this? Well, anytime our index gets bigger than the length of our array, say `N`, we can make a new array of length `N+1`, copy over the `N` values from the old array to the new array, and add our new value being pushed. That idea will certainly work, but it will mean that pushing becomes an $O(N)$ operation, due to the copying of all of these values. So that would be a pretty terrible implementation.

Maybe we can be a bit more clever by adding not just one more cell to our array but 1,000 or 1,000,000 cells to our array. This will mean that only some of the time we have to copy of our array and incur the $O(N)$ cost, but we're still at $O(N)$ in the worst case, which is pretty bad.

We can really solve this problem using the import idea of **array doubling**: anytime our underlying array gets full, we create a new array of *double* the size and add on our value being pushed. We can implement this in code for our older data structure as follows.

```
def create(size):
    A = [''] * 1 # Can ignore size now
    top = 0
    return (1, top, A)

def push(v, stack):
    (size, top, A) = stack
    if top == size:
        tmp = [''] * (2 * size)
        for i in range(top):
            tmp[i] = A[i]
        A = tmp
    A[top] = v
    top += 1
    return (2*size, top, A)

def pop(stack):
    (size, top, A) = stack
    top -= 1
    v = A[top]
    new_stack = (size, top, A)
    return (v, new_stack)

def is_empty(stack):
    (size, top, A) = stack
    return top == 0
```

Now `create()`, `pop()`, and `is_empty()` are in $O(1)$ – great. But `push()` is still in $O(N)$ in the worst case – when we need to double our array – so it doesn't seem like we've really achieved much.

But let's try to be a bit more discerning here. We've discussed that worst-case analysis is often the right measure of complexity, but this is not *always* true. Sometimes it is in fact better to consider a notion of the average complexity of an operation, and this is such a case.[14] When pushing repeatedly onto our stack, we will only sometimes have to double our array, and the doubling will in fact grow increasingly rare as our array grows (can you see why?).

We really want to analyze our data structure while keeping in mind that it will be used over a sequence of operations, not just once. This idea, of studying the cost associated to a sequence of calls rather than a single call, is known as **amortized analysis**. In this case, when pushing $N = 2^k$ elements onto our stack, we will incur a total array doubling cost of $1 + 2 + 2^2 + 2^3 + \cdots + 2^{k-1} = 2^k - 1$. Over the course of $N = 2^k$ calls to `push()`, that comes out to an average cost of $\frac{2^k-1}{2^k} < 1$ due to array doubling. So our amortized analysis tells us that pushing is actually $O(1)$ with array doubling when amortized over a sequence of calls.

Next time we'll continue discussing data types and data structures, and begin learning about object-oriented programming.

§23 Friday, March 25

§23.1 Recap

We're going to keep talking about data types and data structures, and – as promised – we'll begin discussing object-oriented programming (OOP). A quick warning: googling "object-oriented programming" will give you tons of results about software engineering and advanced OOP ideas, but you really shouldn't pay much mind to those for now. The goal in CS1 is really to convince you that OOP is something natural and useful to have, and that it will make our life quite a bit easier.

A quick recap before diving into things: we've been talking about this idea of a stack data type, and thanks to the stack Monday's problem was fairly easy to solve. Without the stack, the problem was difficult even to approach, but making use of the concept makes the problem fairly intuitive. The key idea to the stack was that we have four operations. We can:

1. Create a stack.
2. Test whether our stack is empty.
3. Push an element onto our stack.
4. Pop an element from our stack.

We also talked about 3 implementations for the stack (or *data structures*, using the language we recently learned).

1. Array + index.
2. Nested tuples.
3. Resizeable array + index + size.

[14] I.e., we will consider the average cost of the operation over a sequence of calls, rather than the average over all possible inputs of size N.

As always, we followed each implementation with an analysis of its complexity, i.e., of its runtime behavior. In the case of the resizeable array, we used *amortized analysis* to see that pushing to such an array has an amortized cost of O(1).

Recall also that, after writing functions for creating, pushing, popping, and testing emptiness, we wrote the following solution for the parenthesization problem.

```
 1  def solution(s):
 2      N = len(s)
 3      stack = create(N)
 4
 5      for i in range(N):
 6          c = s[i]
 7          if c == '(' or c == '{' or c == '[':
 8              stack = push(c, stack)
 9          else:
10              if is_empty(stack):
11                  return False
12              (v_top, stack) = pop(stack)
13              if not match(v_top, c):
14                  return False
15
16      return is_empty(stack)
```

But there's something a bit funny going on here – what exactly is the `stack` variable on line 3? We're calling it `stack`, and we'd like to think of it as an instance of the stack data type, but it's really something quite a bit more concrete. In fact, it's just a tuple of several values, which you could unpack and even directly modify if you wanted to.

We'd instead prefer to keep things a bit cleaner and more abstract. To see precisely what we mean, let's pivot a bit and think about numbers. What exactly *is* a number? How do we define a number, and what is the essence of a number? One thing you might say is that a number is a decimal expansion like 2.31, and that you can add them, subtract them from one another, etc. We also learn that they not only have operations and representations, but also properties. For instance, $x + (y + z) = (x + y) + z$ always, which is known as *associativity*.

So, is that a number? What if we just drew four many dots on a piece of paper? You might reasonably think of that as representing the same thing as 4, and IV is yet another example! So what's really going on here – how can we capture the essence of what it means to be a number? Really, a number is abstractly any element of a set S with operations for addition and multiplication satisfying the nice properties that we're used to (e.g., associativity, etc.).

Our discussion of the stack is similar; you shouldn't need to worry at all about how the stack is represented, you should only have access to its operations and their nice properties. In fact, you shouldn't even be able to see the underlying implementation of the abstract data type!

As a first approximation, that's what object-oriented programming (OOP) is about. We have an abstract notion, with a couple key functionalities and many possible implementations/representations, but we want to keep the implementation details out of sight (and sometimes out of mind, as well). Simply put, when it comes time to use a data type, we shouldn't need to – or even be able to! – see the underlying data structure. This is where OOP comes in.

§23.2 Object-oriented programming

The bread and butter of object-oriented programming (OOP) is a **class**, which is roughly a way of packaging a data structure into a nice interface (more akin to a data type). Here's the syntax.

```
class Stack:
    def __init__(self,):
        self.A = [0]
        self.top = 0
        self.size = 1
```

A point on notation: functions inside classes are called **methods**, while data/values are called **attributes**. We've started off by giving our class a special initialization method `__init__()`, which lets Python know that this is how new objects of our class are created. More generally, methods that start and end with `__` allow Python to do some magic with syntax. In this case, we can now write the following:

```
stack = Stack()
type(stack) # evaluates to __main__.Stack
```

Furthermore, we can make the `self.A`, `self.top`, and `self.size` attributes protected by starting their names with two underscores. This will mean that users of the `Stack` class are not allowed to access these underlying attributes. Let's make that change and keep working on our class.

```
class Stack:
    def __init__(self,):
        self.__A = [0]
        self.__top = 0
        self.__size = 1

    def push(self, v):
        if self.__top == self.__size:
            self.__size *= 2
            tmp = [0] * self.__size
            for i in range(self.__top):
                tmp[i] = self.__A[i]
            self.__A = tmp
        self.__A[self.__top] = v
        self.__top += 1
```

Now we have a method for pushing onto a stack, implemented using array doubling, and its syntax is a bit different than what we're used to.

```
stack = Stack()
stack.push(3)
```

Now we can begin to see what the `self` parameter is doing – it's representing the object of the class on which the method is being used![15] Now let's add a parameter for popping. We'll add the following code to our class.

```
    def pop(self):
        self.__top -= 1
        return self.__A[self.__top]
```

And if a user tries to be malicious and access the values `self.__top` or `self.__A`, they'll run into an `AttributeError`, since the underscores tell Python to protect the attribute. For example,

```
stack = Stack()
stack.__top
```

will throw an `AttributeError`.

Something to note: we're not writing `stack = stack.push(3)`, we're just writing `stack.push(3)`. That's because these methods don't return anything; they're actually modifying objects in-place. So `stack.push(3)` changes the object `stack` itself, and we can move forward knowing that a value has indeed been pushed onto `stack`.

Using this new OOP syntax, our previous solution to the parenthesization problem would look something like this.

```
def solution(s):
    N = len(S)
    stack = Stack()

    for i in range(N):
        c = s[i]
        if c == '(' or c == '{' or c == '[':
            stack.push(c)
        else:
            if len(stack) == 0:
                return False
            v_top = stack.pop()
            if not match(v_top, c):
                return False

    return len(stack) == 0
```

[15] The jargon is piling up a bit, so feel free to take a minute to let things sink in.

What was the point of all this? Our code is now more abstract, easier to read and reason about, and more modular. We'll keep playing with these ideas in the coming weeks, and hopefully you'll soon agree that OOP is a useful way of structuring and reasoning about code. Enjoy the weekend!

§24 Monday, March 28

§24.1 Classes, continued

Last time we created a class to implement the stack type, with both attributes and methods. The attributes included the underlying array of the stack, an index telling us where we are in the array, and the length of that array. We also had methods for implementing the stack properties (like pushing and popping).

Remark 24.1. In Python, absolutely everything is an object from a class, even the simplest things like `int` s and `bool` s!

Example 24.2

Let's say we want to define the complex numbers in Python using a class. We might write something like:

```
class Complex:
    def __init__(self, r, i):
        self.real = r
        self.imaginary = i
```

This gives us a nicer interface for complex numbers than just storing them as tuples of floats, for instance. (And we can then implement addition, subtraction, multiplication, etc. if we'd like.)

So, in Python everything is an object. If we write something like `x = 3`, what exactly is happening? What happens is that `x` is created as a reference to the value `3`, which is stored somewhere in memory. In fact, we could even create an object with two parts: an integer and a reference to another object (which itself is an integer and a reference to another object, etc.). This kind of idea can allow us to implement **linked data structures**. Let's pursue this idea further.

```
class Node:
    def __init__(self, v, tl):
        self.value = v
        self.tail = tl
```

Now we have a simple class for nodes, which are the building blocks of linked data structures. We can now implement a stack as a linked data structure as follows.

```python
class Stack:
    def __init__(self):
        self.head = None
        self.size = 0

    def push(self, v):
        new_node = Node(v, self.head)
        self.head = new_node
        self.size += 1

    def pop(self):
        v = self.head.value
        self.head = self.head.tail
        self.size -= 1
        return v

    def __len__(self):
        return self.size
```

It's even possible to take this idea further and create nodes with left tail and right tail attributes (which give rise to the tree data structure), etc.

§24.2 More types!

Let's talk a step back here. So far we've mentioned a handful of types with certain essential operations (or methods, once we set them up as classes).

- Stack: push, pop (last in & first out, or LIFO)
- Queue: enqueue, dequeue (first in & first out, or FIFO)
- List
- Set
- Dictionary

We haven't spoken too much about the last couple types, but they're very important. In fact, they're so important that they're built right into Python!

§24.2.1 List

A **list** is like an array where you're also allowed to add and remove elements. (That is, the length of a given list can change over the course of its lifetime). Here's the syntax of using lists.

```python
l = [2, 3, 4, 5]
l[2] # is 4
```

```python
l[2] = 56
l # is [2, 3, 56, 5]
```

Now let's show off that lists can actually grow, unlike arrays. The syntax to add an element to the end of `l` (or to **append** to `l`) is as follows.

```python
l.append(567)
l # is [2, 3, 56, 5, 567]
```

It's also important to keep track of the complexity of these list operations. This `site` has detail on the runtime behaviors of these operations – you should be able to make sense of them, keeping in mind that Python implements lists using the dynamic array doubling that we mentioned previously.

§24.2.2 Set

A **set** is a type for storing a collection of items where you can add items to the collection, remove item from the collection, and test membership of a value in the collection. Informally, sets are like backpacks: you can place values in them (without any notion of order), pull values out, and check whether something lives in your backpack. One additional detail is that sets don't permit repetition.

In this sense, they're like sets from mathematics; they simply keep track of the distinct things you have, they don't keep track of repeated elements. (For instance, to a mathematician the sets $\{2, 3, 4\}$ and $\{2, 4, 3, 4\}$ are identical.)

§24.2.3 Dictionary

The **dictionary** is a type that stores (key, value) pairs. This supports insertion of (key, value) pairs and lookup of the value associated to a given key. For instance, we might write:

```python
d = dict()
d['Joe'] = 'B-'
d['Caleb'] = 'A'
'Joe' in d # evaluates to True
d['Joe'] # evaluates to 'B-'
```

Next time we'll talk about the complexity of these operations for sets and dictionaries, which will involve understanding a bit about how they work under the hood.

§25 Wednesday, March 30

§25.1 Dictionary, continued

Last time we introduced a couple new players: lists, sets, and dictionaries. We mentioned that a dictionary is a collection of (key, value) pairs and that we usually expect at least three functionalities from a dictionary:

- Insertion of a (key, value) pair into a dictionary.

- Look-up of the value corresponding to a given key (which must be unique!).
- Testing the existence of a key in the dictionary, i.e., whether there exists a (key, value) pair where key=`k` for given `k`.

Dictionaries are a common and powerful data type, and today we'll discuss how to actually implement them. We'll kick things off with a class for (key, value) pairs (or KVPs).

```
class KVP:
    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __str__(self):
        return str(self.key) + ' : ' + str(self.value)
```

Great – now we can write things like `kv = KVP('Joe', 'B-')`. And because we've implemented a `__str__()` method, Python now knows how to turn KVPs into strings, which allows it print them! Now comes the hard part: actually implementing the dictionary data type. One reasonable place to start is to use a `list` as the underlying data structure for a dictionary, where each entry has some KVP. Try to think about how you would implement insertion, look-up, and testing if you were to use a `list` to implement the dictionary.

Now let's get to writing (we'll use `self.repr` as shorthand for `self.representation`).

```
1 class LDict:
2     def __init__(self):
3         self.repr = []
4
5     def __contains__(self, key):
6         for kv in self.repr:
7             if kv.key == key:
8                 return True
9         return False
```

Okay, now we can create empty dictionaries and lookup keys in our dictionaries. The magic method `__contains__()` allows us to write `k in my_dict` where `my_dict = LDict()`. Simply put, Python is thoughtful enough to read `k in my_dict` and then do `my_dict.__contains__(k)`, which has the effect of making things more readable for us humans. Furthermore, in line 6 we're making use of the fact that we can iterate directly over lists. For instance, if `A` is a `list` type in Python then we can write

```
for a in A:
    # do something with a
```

rather than

```
for i in range(len(A)):
    a = A[i]
    # do something with a
```

which is easier to read and write. Okay, now we need a method for inserting values into a dictionary. Rather than writing something like `my_dict.insert_kvp(key, value)`, it would be nice if we could just write `my_dict[key] = value`. Fortunately Python allows us to do just that, by making use of the magic method `__setitem__()`.

```
class LDict:
    def __init__(self):
        self.repr = []

    def __contains__(self, key):
        for kv in self.repr:
            if kv.key == key:
                return True
        return False

    def __setitem__(self, key, value):
        kv = KVP(key, value)
        self.repr.append(kv)
```

Now all that's left is lookup. In order to use the notation `my_dict[k]` rather than something like `my_dict.get_value(k)`, we'll use the magic method `__getitem__()` this time. Our first prototype is now complete.

```
class LDict:
    def __init__(self):
        self.repr = []

    def __contains__(self, key):
        for kv in self.repr:
            if kv.key == key:
                return True
        return False

    def __setitem__(self, key, value):
        kv = KVP(key, value)
        self.repr.append(kv)

    def __getitem__(self, key):
        for kv in self.repr:
            if kv.key == key:
                return kv.value
```

As always, we now need to think about the complexity of these operations. Creation is certainly $O(1)$, and insertion is just an append operation on a `list` type, which we saw is amortized $O(1)$. Lookup and testing both involve iterating over `self.repr`, so they are $O(N)$ in the worst case. That's not a very good implementation at all: $O(N)$ lookup and testing are quite bad if we want to work freely with dictionaries containing millions or billions of entries.

How can we speed things up? One way of speeding up a search that we've already seen is to use binary search! In this case, that would require for us to keep our KVPs sorted by their keys. We can do that, so let's think about how that changes things. If we were to use a sorted list as the underlying data structure, then insertion would be $O(N)$ in the worst case, as we might need to move all the entries in our array by 1 in order to fit our new KVP in the right spot. The payoff is that, using binary search, lookup and testing would both be $O(\log N)$ in the worst case. So there's a tradeoff here – insertion gets slower but lookup and testing get faster.

It turns out that there's a way of doing things so that we won't need to make *any* compromises: by using a **hash table**. Hash tables are pretty magical, and they rely on the idea of generalizing an array with integer indices to an array with arbitrary indices (roughly speaking). More explicitly, say we had a magical function `h()` for mapping keys to integers such that different keys always map to different integers.

Then we could simply use a big array `A` as our underlying data structure for dictionaries where `A[h(k)] = v` for each KVP (k, v) in our dictionary. Take a moment to understand why this makes sense, and why it would make insertion and lookup be $O(1)$! Simply put, such an `h()` would solve all of our problems, and is known as a **hash function**.

Right now, this is a romanticization for several reasons:

1. In practice, `h()` might give rise to a **collision**, i.e., `h(k1) == h(k2)` for distinct keys `k1` and `k2`. This causes a problem with how we store `(k1, v1)` and `(k2, v2)` in our underlying array `A`. (They can't both go to `A[h(k1)]`!).

2. How big does our underlying array `A` need to be? What if `h()` can spit out integers between -1e10 and 1e10, but we only need to store a couple thousand KVPs? We wouldn't want to lug around a huge array in memory anytime we're using a dictionary.

It turns out that point (1.) can be addressed using something called *chaining* and point (2.) can be addressed by setting `A` to be of size `N` and replacing `h(k)` with `h(k) % N`. More on dictionaries in the coming lectures.

§26 Friday, April 1

A couple quick administrative notes: after the next midterm, we'll be discussing how to make a video game in Python! We've already covered lots of ground concerning Python fundamentals and algorithmic thinking, so we'll do something a bit more fun for the end of the course. (And it will also teach us a lot about using libraries, organizing larger pieces of code than we're used to, etc.)

§26.1 Problem: rings and rods

Here's a cool problem, from that nice website LeetCode that we mentioned earlier: rings and rods. The key idea is that for each rod, we want to keep track of the rings that are placed on it, but we don't actually care whether a rod has 5 red rings or 10 red rings. We only need to keep track of whether a given rod does or does not have a certain ring color, and doing so will make sure our code runs faster (e.g., we won't be slowed down by a rod that has millions of red rings).

So we want to keep track of the rings on a given rod in a way that doesn't allow for repetitions, and we also don't care about the order of the rings on a given rod. That sounds a lot like we want to use the `set` data type! Also, we'll need to keep track of this collection of rings for each given rod, which means a `dict` will be handy as well. Let's start writing a solution.

```python
class Solution:

    def countPoints(self, rings: str) -> int:
        rods = {}
        for i in range(10):
            rods[i] = set()

        N = len(rings) // 2
        for i in range(N):
            color = rings[2*i]
            rod = int(rings[2*i + 1])
            rods[rod].add(color)

        count = 0
        for i in range(10):
            if len(rods[i]) == 3:
                count += 1

        return count
```

Nice! We can check that this solution works in LeetCode, which will in fact tell us that it's faster than most submitted Python solutions. Hopefully it's satisfying to see techniques that we've learned come together to form simple and efficient solutions.

§26.2 Semantics

Let's walk through some Python code using python tutor. If we run the following code, what exactly is happening?

```python
class Test:
    def __init__(self):
        self.n = 42
```

```
4
5       def f(self, x):
6           y = x + self.n
7           return y
8
9   o = Test()
10  o.f(21)
```

After line 8, Python has been informed of the existence of a class `Test` with two methods. There aren't any objects in this class yet, but Python has been taught how to work with the class if it is instructed to. In line 10, an object `o` is created with two methods and an attribute (can you see what they are?). And in line 11, a method of `o` is called (`o.f()`), which itself makes use of one of `o`'s attributes (`o.n`).

Python tutor can be a great resource for understanding the meaning (or *semantics*) of Python code, so feel free to play around with it yourself! Let's look at another, trickier example.

```
1   def f(x):
2       y = x + 1
3       return y
4
5   def g(x):
6       y = f(x) + 4
7       return y
8
9   def h(x):
10      y = g(x) + f(x)
11      return y
12
13  result = h(4)
```

This is a hard one to explain on paper, but it's worth following carefully in python tutor. One of the key lessons is that functions have their own environments. For instance, the `x` in `f(x)` on line 10 and the `x` on line 2 are *not* the same thing. They may take the same value at some moments, but they are allowed to vary.

Another important observation is that the manner in which these functions are executed follows a familiar format: that of a stack! In particular, line 13 involves computing `h(4)`, which is now the first order of business, so to speak (i.e., the topmost element of our stack). But `h(4)` makes use of `g(4)` and `f(4)` so executing those computations is now the first order of business (i.e., they are now at the top of our stack). And, finally, when calling `g(4)` we are then forced to compute `f(4)`, which now becomes the first order of business. This is precisely a stack! Simply put: if the first order of business is to compute `f1(x)` but `f1(x)` uses `f2(x)`, then computing `f2(x)` now becomes the first order of business. (And once `f2(x)` has been evaluated, we can proceed with our original goal of computing `f1(x)`, which becomes the highest priority.)

We can take this one step further: rather than having functions use other functions, we can have a function use itself! This idea is known as **recursion**, and it's one of the most important concepts in computer science. Let's take it for a spin.

```
def f(x):
    y = f(x + 1)
    return y

f(1)
```

Cool, we have a function `f()` that makes use of itself – or *recurs* – and we'll test it on input `1`. Let's run it. Uh oh, we've run into a `RecursionError`, as the maximum recursion depth was exceeded.

What happened here? In trying to execute `f(1)`, our program called `f(2)`, which then called `f(3)`, which called `f(4)`, and so on. This process will never end, but Python is smart enough to eventually call it quits and warn you that your function is trying to call itself too many times (thus giving us our `RecursionError`).

Here's an example of a recursive function that *does* work (at least some of the time). Try to see if you can understand why.

```
def f(x):
    if x == 3:
        return 42
    else:
        return f(x + 1)

f(1) # evaluates to 42
```

§27 Monday, April 4

An administrative note: please organize into groups of size ≤ 4 for the final project by this Friday. This can be done under People/Groups on Canvas. Furthermore, there won't be any discussion sections next week or the following week due to the time off for Easter. The TA's will instead be holding office hours for homework and midterm prep during their usual discussion times, and you can feel free to attend any of these office hours.

Lastly (and also due to Easter weekend), HW9 will be due on Wednesday of next week, and it'll be only 2 problems rather than the usual 4 or 5.

§27.1 Lists operations

Before diving into functional programming and dynamic programming, let's briefly revisit lists. One thing to note is that there are essentially two kinds of operations on lists: those that are in-place and those that are not. More explicitly, there are operations that change the list `A` on which they are called and those that return a different list without altering `A`.

For instance, assignment is in-place, meaning the following code changes `l` itself.

```
l = [2, 5, 3, 6, 7, 3, 4]
l[2] = 42
l # evaluates to [2, 5, 42, 6, 7, 3, 4]
```

Slicing, meanwhile, is not in-place. Writing something like `l[2:5]` would not change `l` but rather create another list entirely. This duality can be important to keep in mind when writing programs with lists.

§27.2 Problem: sum of a list

Here's a problem: given a list of integers, compute the sum of the entries in the list. This is much easier than some of the problems you've had to solve on the homework and midterms, and you might even have an idea of how to solve it right off the top of your head. In particular, we can just walk through the elements of `l` and add each of its entries to an accumulator variable `acc`.

Let's write that up.

```
def lsum1(l):
    acc = 0
    for i in range(len(l)):
        acc += l[i]
    return acc
```

This is indeed a correct solution, but – just for the fun of it – let's try to think about other possible solutions. One technique we've seen is that of divide & conquer, which certainly applies here. Let's write up a divide and conquer solution using recursion.

```
def lsum2(l):
    if l == []:
        return 0
    else:
        # Divide
        v = l[0]
        subproblem = l[1:]
        # Solve subproblem "magically"
        solved_subproblem = lsum2(subproblem)
        # Conquer
        answer = v + solved_subproblem
        # Done
        return answer
```

To really illustrate what's going on here, let's write out the steps of computation in executing `lsum2([2, 3, 4])`.

```
lsum2([2,3,4]) = 2 + lsum2([3, 4])
               = 2 + (3 + lsum2([4]))
               = 2 + (3 + (4 + lsum2([])))
               = 2 + (3 + (4 + 0))
               = 2 + (3 + 4)
               = 2 + 7
               = 9
```

In fact, now that we've learned about recursion we can write up other divide & conquer algorithms that we know and love. Insertion sort is a good place to start.

```python
# l is already sorted
def insertion(v, l):
    if l == []:
        return [v]
    elif l[0] >= v:
        return [v] + l
    else:
        return l[:1] + insertion(v, l[1:])

def insertionSort(l):
    if len(l) <= 1:
        return l
    else:
        # Divide
        v = l[0]
        subproblem = l[1:]
        # "Magic"
        solved_subproblem = insertionSort(subproblem)
        # Conquer
        answer = insertion(v, solved_suproblem)
        # Done
        return answer
```

To be clear, we're being very verbose with comments and variables to make things as clear as possible. In 'real life', you'd probably instead see something like this.

```python
def insertionSort(l):
    if len(l) <= 1:
        return l
    else:
        return insertion(l[0], insertionSort(l[1:]))
```

So, what's the advantage of the recursive approach to solving algorithmic problems? For starters, some people find this style of problem-solving/coding much easier than other styles. In some countries, like France, this recursive approach to programming is actually how students are taught from high-school. More abstractly, recursion often makes seemingly intractable problems much more approachable.

A final point is that these kinds of recursive solutions can also make the parallelization of a computational task across several devices much easier than other solutions. For this reason, recursion is sometimes preferred in industrial applications where inputs to programs can easily have billions of entries or more. (And perhaps most importantly, *understanding* recursion will make you a stronger and more versatile computer scientist.)

Index