CCC Junior Preparation Book

By: Alan Bui

July 21, 2023

Introduction

This was created for the Vincent Massey Computer Science Club Junior Stream for the 2022-23 school year. Special thanks to Amanbir for helping write some parts of the lesson ~~during Writer's Craft~~.

Some important resources before we start:

DMOJ will be used for practice problems.
Replit is a versatile online IDE and I suggest to begin with because it is simple, online, and you can use it on multiple devices.

Later on, if you want to install Python for your own computer, this might help: https://www.youtube.com/watch?v=oaZ4PXqUO9c.

Table of Contents

Lesson 1 - Introduction

In this document, we will discuss the fundamentals of competitive programming using Python. We're going to be doing a lot of pressing random keys on the keyboard and hoping it works... I mean programming!

Variables

Computer science uses variables and you can name them pretty much whatever you want with a few exceptions such as not starting with a number, being a keyword, or having spaces in the name. If you can, try to keep your variable names reflective of what it is doing (name it based on what it does).

Variables can have different types; here are the main ones:

Integer - Numbers with no decimal point (e.g. 1, 2, 907421)
Float - Numbers with a decimal point (e.g. 3.1, 2.7539, 1.0)
String - Characters surrounded in quotations (e.g. "abc", '123', '')
Boolean - it's either True or False (e.g. True, False)

To assign a variable a value, we use the following syntax:
variable_name = whatever you want the variable to be
For example:

```
Name = "Alan"
Age = 15
Teacher = True
chocolateEaten = 4.5
```

Operations

Just like a calculator, there are some operations Python can do:
Addition (+)
Subtraction (-)
Multiplication (*)
Exponent (**)

Division (/)
Integer Division (//) - floors the answer to the division operation
Modulo (%) - Gives the remainder of the division operation

floor(x): returns the largest integer less than or equal to x
floor(2.9) = 2
floor(-2.9) = -3

```
>>> 3+2
5
>>> 3-2
1
>>> 3*2
6
>>> 3**2
9
>>> 3/2
1.5
>>> 3//2
1
>>> 3%2
1
```

Printing values

You can output values using the print() keyword

For example:

```
print("Hello, World!")
```

You can output all kinds of values like strings, numbers, variables, and much more we will learn in the future! By default, print() will add a newline to the end of whatever it is you are outputting.

For example,

```
print(1)
print(2)
```

will output:

```
1
2
```

*Notice how the numbers are on different lines

If you want things separated on the same line, you can use a comma.

```
print(1, 2)
```

will output:

```
1 2
```

Taking input

You can take user input using the keyword input()

For example:
`x = input()`
This will prompt the user for an input and store it in the variable x. The default type for input is string.

Changing Types
If you want to change the variable x to be a different type, you can use the int(), float(), str(), and more keywords.

Ex.

```
x = int(x)
x = float(x)
x = str(x)
x = bool(x)
```

Comments:

If you have stuff with # in front of the line, then you can see it but the Python interpreter cannot.

Ex.

```
#This is a comment
print("Hello") #This is a comment too
```

DMOJ Problems:

https://dmoj.ca/problem/helloworld
https://dmoj.ca/problem/ccc13j1
https://dmoj.ca/problem/ccc22j1

Editorials can be found here

Lesson 2 - If logic

If Statements

All if statements are of the form:

```
if (condition is True):
     bunch of stuff
```

Operators you can use:

```
> #greater than
< #less than
>= #greater than or equal to
<= #less than or equal to
== #equal to
!=
```

For example:

```
name = "Alan"
if name == "Alan":
     print("HI ALAN")
```

Notice the use of “==” instead of “=”. One equal sign is to assign variables. Two equal signs are used to compare.

Else statements

You can use else in combination with an if statement.

```
if (condition is True):
     #Bunch of stuff
else: #This code ONLY runs when the condition is False
     #Bunch of stuff
```

For example:

```
print("What mark did you get?")
mark = input()
mark = int(mark)

if mark < 50:
     print("YOU FAILED")
else:
     print("YOU PASSED")
```

Combining else and if:

If you have an if statement inside of an else statement, you can substitute that for elif (short form for else if). Note that this only works in python.

For example this:

```
mark = 30

if mark < 50:
     print("Fail")
else:
     if mark < 100:
          print("Bad")
     else:
          print("Pass")
```

Could be written as:

```
mark = 30

if mark < 50:
     print("Fail")
elif mark < 100:
     print("Bad")
else:
     print("Pass")
```

and

You can use the and keyword for multiple conditions to be True.

```
name = "Isaac"
wearsGlasses = "False"

if name == "Isaac" and wearsGlasses == "False":
     print("THIS IS ISAAC")
else:
     print("THIS IS AN IMPOSTER")
```

or

You can use the or keyword for if either of the conditions are True

```
name = "Isaac"
wearsGlasses = "False"

if name == "Isaac" or wearsGlasses == "False":
     print("THIS IS MAYBE ISAAC")
else:
     print("THIS IS DEFINITELY AN IMPOSTER")
```

DMOJ Problems:

https://dmoj.ca/problem/gfssoc3j1
https://dmoj.ca/problem/ccc08j1
https://dmoj.ca/problem/ccc20j1
https://dmoj.ca/problem/ccc17j1
https://dmoj.ca/problem/ccc21j1
https://dmoj.ca/problem/ccc18j1
https://dmoj.ca/problem/ccc07j1

Editorials can be found here

Lesson 3 - Loops

Loops (Repetition)

Sometimes in programming, we want to do some task multiple times. This is very useful for us because we can take advantage of the computer's speed/power to do calculations over and over again. Also, if we want to run some code (for example, taking user input) multiple times we won't have to copy paste lines and lines of code.

While loops

While loops are of the form:

```
while (condition is True):
     #bunch of stuff
```

As long as the condition is True, the program will keep executing the code inside the indented block. The condition is checked at the start of each loop iteration and only there.

Example:

```
x = 1
while x < 4:
     x = x + 1
     print(x)
```

You will notice that x = x + 1 eventually causes x to become 4, which makes the condition false.

Woah, what happened here? How is it possible to do x = x + 1??? Mathematically, that makes no sense. That is true. But, remember in computer science, we have variable assignments in the form: Variable_name = bunch of stuff

In this case, we are creating a variable x with the value of x+1, where x+1 is 1 greater than the previous value of x. Another way of writing this is:

```
x += 1
```

This code increases the value stored in the variable x by 1.

A different condition:

```
Name = ""
while Name != "done":
     print("Enter your name or enter done to exit")
     Name = input()
```

Here we will demonstrate a while True loop!

```
while True:
     print("What mark did you get?")
     mark = int(input())
     if mark == 100:
          print("Good job!")
          break
```

Doing while True will loop indefinitely because True is always True. It's just like saying while 1 == 1. Notice the use of the **break** keyword. When your code "breaks" it exits the loop it's currently inside of. This is VERY IMPORTANT to avoid infinite loops. An infinite loop occurs when the loop goes on and on forever because there is no way to get out of the loop.

For example:

```
while 1 == 1:
     print("HELLO")
```

Since 1 is always equal to 1, this code will go forever and ever and ever. Always make sure that when you create a while loop, there is **some condition that causes the loop to end**.

For Loops

For loops are generally of the following form:

```
for variable_name in range(start, stop, step):
     #bunch of stuff
```

Here, the range keyword will start at "**start**", stop up to but NOT INCLUDING "**stop**" and increase by "**step**".

The default value of start is 0 and the default value of step is 1.

```
for i in range(1, 11, 2):
     print("Hello", i)
```

This will start i at 1 and increase by 2 each time the loop iterates as long as i is less than 11. This is a counted loop and is used when you know how many times you want to do something. For example, if a problem asks you to take 5 numbers as input and output the average, you can do:

```
avg = 0
for i in range(5):
     num = int(input())
     avg += num
print(avg/5)
```

DMOJ Problems

If statement review:
https://dmoj.ca/problem/ccc12j1
https://dmoj.ca/problem/vmss7wc15c1p1

Using a loop:
https://dmoj.ca/problem/valentines19j1
https://dmoj.ca/problem/acmtryouts0a
https://dmoj.ca/problem/ccc21j2
https://dmoj.ca/problem/ccc20j2

Editorials can be found here

Lesson 4 - Lists and Strings

Lists

What is a list? Why would you ever need it? Let’s say you had 10 students whose ages are 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10. How would you represent this?

You could do:

```
age1 = 1
age2 = 2
age3 = 3
age4 = 4
... and so on
```

BUT that’s really tedious. If I asked you for the sum of the first 5 people, it would be a lot of work. Instead, we can represent these ages as a list, denoted by [] square brackets.

Lists are created in the form

```
variable_name = [] #creates an empty list
```

So, here we could do:

```
ages =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Alternatively, we could use the append() keyword. What this does is adds whatever element to the end of the list.

```
ages = []
for age in range(1, 11, 1):
     ages.append(age)
```

To access elements in the list, we use:

```
list_name[index]
```

List indices start at 0. So here, ages[0] = 1, ages[1] = 2, ages[2] = 3, etc. This is a very common mistake on ICS3U tracing tests/assignments. I am totally not still salty about losing marks over this. So be careful!

Useful keywords:

list_name[index]	Points to the element of the list at index (counting from 0)
list_name[first:last]	The elements of the list starting at first and going up to but NOT including last (counting from 0)
list_name.append()	Puts whatever is in the brackets to the end of the list
list_name.remove()	Removes the first instance of whatever is in the brackets
del list_name[index]	Deletes the element in the list at index (counting from 0)
list_name.sort()	Sorts the list from smallest to largest
list_name.reverse()	Reverses the list
input().split()	Splits input into a list, split by whatever is in the brackets or “ “ (space) by default
list_name.index()	Returns the first index that equals whatever is in the brackets
len(list_name)	Length of the list
min(list_name)	Returns the smallest element in list_name

max(list_name)	Returns the largest element in list_name

Looping through a list:

```python
nums = [1, 5, 6, 7, 3, 4]
for num in nums: #num becomes each element in nums in order
     print(num)
```

Consider the following:

```python
lis1 = [1, 2, 3, 4]
lis2 = lis1
lis2.append(9)

print(lis1)
print(lis2)
```

Hmmmm, it seems that lis1 updated when lis2 updated. It's for a similar reason as stated earlier. When you do lis2 = lis1, it just takes lis2 and points it to the same spot in memory as lis1. This means when you change lis2, you change the spot in memory that lis1 also looks at! This is referred to as a shallow copy. To avoid this, you want a deep copy which can be accomplished when you do:

```python
lis1 = [1, 2, 3, 4]
lis2 = [i for i in lis1]
```

OR

```python
lis1 = [1, 2, 3, 4]
lis2 = lis1[:]
```

OR

```python
lis1 = [1, 2, 3, 4]
lis2 = lis1.copy()
```

How do we take input here?
https://dmoj.ca/problem/aplusb

We know that if we get the integers “a” and “b” we can easily output their sum by doing:

```
print(a+b)
```

Normally, we are used to taking input with one thing on the line at a time. Unfortunately, we have “a” and “b” on the same line, so using input() twice does not work. This is because input() takes the whole line as input.

If you scroll down, and see the solution, you see:

```
N = int(input()) #takes input for number of cases

for _ in range(N): #loops N times, _ is a variable name (yes, it's
weird)
    a, b = map(int, input().split()) #What is happening here??????
    print(a + b) #outputs the answer
```

Let’s not go down the rabbit hole of what the map is doing. Instead, let’s focus on what input().split() does.

```
names = input().split()
print(names)
```

input().split() will return the input split into a list separated by whatever you put in the split(). By default, it splits by the space.

Back to A+B, this is how I would do the problem:

```
n = int(input())

for i in range(n):
	nums = input().split() #nums becomes a list with two things in it
	a = int(nums[0]) #gets the integer value of the first thing in the list
	b = int(nums[1]) #gets the integer value of the second thing in the list

	print(a+b)
```

Strings

Useful things to do with Strings:

string_name[index]	Points to the element of the string at index (counting from 0)
string_name[first:last]	The elements of the string starting at first and going up to but NOT including last (counting from 0)
string1+string2	= the strings concatenated together E.g. “a”+”b” = “ab”
string_name.count()	Counts the number of times whatever is in the brackets occurs in the string
string_name.find()	Finds the first occurrence of whatever is in the brackets
string_name.rfind()	Finds the last occurrence of whatever is in the brackets

string_name.replace(original, new)	Replaces every occurrence of original with new
string_name.strip()	Removes whitespace from beginning and end of string
string1 in string2	Checks if string1 is inside of string2
len(string_name)	Length of the string
string_name.lower()	Returns string_name in all lowercase letters
string_name.upper()	Returns string_name in all uppercase letters

DMOJ Problems:

Lists:
https://dmoj.ca/problem/a4b1
https://dmoj.ca/problem/bf1hard
https://dmoj.ca/problem/ccc11s2
https://dmoj.ca/problem/ccc03s1
https://dmoj.ca/problem/ccc97s1
https://dmoj.ca/problem/gfsscc21p3

Strings:
https://dmoj.ca/problem/a1
https://dmoj.ca/problem/bfs17p1
https://dmoj.ca/problem/ccc02j2
Editorials can be found here

Lesson 5 - Functions

Before looking into functions, let's look at some ways of taking input.

Two spaced integers 1 2	a, b = map(int, input().split())
List of integers separated by spaces 4 3 5 2 4 6	lis = [int(i) for i in input().split()]

Let's look more into detail on `lis = [int(i) for i in input().split()]`

input().split() is a list
int(i) for i in input().split() sets all i in input().split() to its integer value and puts it in the list called "lis"

This is a shorthand for doing something like:

```
stuff = input().split()
lis = []
for i in stuff:
     lis.append(int(i))
```

Functions

Back to functions. What is a function? As Mr. Ing has so wisely said in my grade 10 math class, "A function is a series of instructions that returns at most one output."

In computer science, it is the same thing. In fact, you may have already used functions without realizing it.

For example:

max(lis) returns the maximum element of the list
lis.sort() returns None... literally nothing

To create your own functions, use the **def** keyword. Functions are of the form:

```
def function_name(parameters):
      #bunch of stuff
```

To call the function in your program just type the function name and the parameters.

For example:

```
def sayHello(name):
      print("Hello", name)

sayHello("ALAN")
```

Outputs:

```
"Hello ALAN"
```

Using the **return** keyword will end the function and return whatever value that follows.

For example:

```
def summation(a, b):
      return a+b

print(summation(1, 2))
```

To return multiple things at once, you can put it in ONE list, so you are technically only returning one thing, just it's all in a list

For example:

```
def reversedList(lis):
      returnedList = []
      for i in range(len(lis)-1, -1, -1):
            returnedList.append(lis[i])

      return returnedList
```

Functions are commonly used to break up your code into different components, then putting everything together. Functions should serve ONE purpose. Another person looking at your code should be able to use the function without having to understand what goes on inside the function. For example, we can use lis.index(element) without really understanding what is actually happening - and that's OK! This means the function was created properly.

Note: Do NOT have 100+ line functions or you will lose marks on Mr. McKenzie's assessments

Local and global variables

```
def change():
     X = 100

X = 50
change()
print(X)
```

The X in the function is considered a local variable. That means, there are two different Xs! The X in the function can ONLY be used within the function. To make the Xs the same, use the **global** keyword.

```
def change():
    global X
    X = 100

X = 50
change()
print(X)
```

Let’s try an example with lists:

```
lis = [1, 2, 3, 4]

def addElement(x, L):
    L.append(x)

addElement(5, lis)
print(lis)
```

What happened here? Lists are passed by reference, which means that L is just pointer that goes to the same spot in memory as lis, so when you append to L, it points to the memory address that lis points to and appends 5.

DMOJ Problems

https://dmoj.ca/problem/collatz
https://dmoj.ca/problem/bfs17p1 (Use a function that returns the value of a word)
https://dmoj.ca/problem/bf3 (Use a function that returns True if the number is prime)

More practice with lists:
https://dmoj.ca/problem/gfsscc21p3

Editorials can be found here

Lesson 6 - Multi-Dimensional Lists

Loops inside of loops

Before we use multidimensional lists, we should understand loops inside of loops, or nested loops.
Example:

```
for i in range(2):
    for k in range(3):
        print(i, k)
```

Let’s trace the program:

i	k	output
0	0	0 0
0	1	0 1
0	2	0 2
1	0	1 0
1	1	1 1
1	2	1 2

Notice how “k” changes before “i”. This is because the inner loop runs first, then the outer loop. Whatever happens in the innermost loops happens first.

Nested loops example: https://dmoj.ca/problem/ccc97s1

Solution:

```
def solve():
    num_subjects = int(input())
    num_verbs = int(input())
    num_objects = int(input())

    subjects = []
    verbs = []
    objects = []

    for i in range(num_subjects):
        word = input()
        subjects.append(word)

    for i in range(num_verbs):
        word = input()
        verbs.append(word)

    for i in range(num_objects):
        word = input()
        objects.append(word)

#finish in class
    for subject in subjects:
        for verb in verbs:
            for object in objects:
                print(subject, verb, object+".")


n = int(input())
for i in range(n):
    solve()
```

Lists inside of lists

Just like loops inside of loops, we can have lists inside of lists. This is referred to as multidimensional lists. When you make a list, you use the same syntax with the square brackets, just you can put lists inside of those square brackets (even more square brackets!)

Example:

```
twoDimensionalList = [[1, 2, 3], [4, 5], ["6", 7, 8, 9]]
```

For a better visual representation:

```
twoDimensionalList = [
    [1, 2, 3],
    [4, 5],
    ["6", 7, 8, 9]
]
```

Notice how the inner lists do not necessarily have to be the same size or store the same types.

You can even put lists inside of lists inside of lists (3D-lists)! But, there are not as many applications of this.

Example:

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

You can even have a 4D-list, but I'm not so sure what that would represent in the real world.

We will stick with learning 2D-lists, but the extension to nD-lists is not far off.

Accessing elements in a 2D-list

To access elements, we will go from broad, to more specific. That is, first we want to access the list inside of the list. We are still going to use square brackets for accessing list indices. Remember 0-indexing!

Example:

```
twoDimensionalList = [
    [1, 2, 3],
    [4, 5],
    ["6", 7, 8, 9]
]
innerList = twoDimensionalList[0]
print(innerList)
```

Once we have access to the innerList, we want to access the specific elements in the innerList. This is just like using 1-Dimensional lists, as we did previously.

```
twoDimensionalList = [
    [1, 2, 3],
    [4, 5],
    ["6", 7, 8, 9]
]
innerList = twoDimensionalList[0]
print(innerList[1])

#This can also be done in one line
print(innerList[0][1])
```

Once we have the index of the element, changing its value is just like any other variable

List comprehension

Let’s make a list filled with a bunch of 0s.
Using .append():

```
twoDimensionalList = [[], [], []]
for i in range(3):
    for k in range(2):
        twoDimensionalList[i].append(0)

print(twoDimensionalList)
```

Using list comprehension:

```
twoDimensionalList = [[0 for i in range(2)] for k in range(3)]
print(twoDimensionalList)
```

[0 for i in range(2)] is a list with two 0s -> [0, 0]
for k in range(3) does it three times
The [] on the outside makes it a list

Both will output the 2D-list.

```
[[0, 0], [0, 0], [0, 0]]
#OR with better visual representation
[[0, 0],
 [0, 0],
 [0, 0]]
```

DMOJ Problems

https://dmoj.ca/problem/ppwindsor18p2
https://dmoj.ca/problem/hkccc15j3
https://dmoj.ca/problem/gfssoc1j4
https://dmoj.ca/problem/dwite08c4p2
https://dmoj.ca/problem/ccc18s2

Solutions can be found here

Lesson 7 - Data Structures

Libraries

Just like how you don’t bring your house to school everyday, Python does not bring every class/function whenever you run Python code. You can import libraries using the **import** keyword.

Queues

Queues are just like line-ups. In fact, in some places (such as Australia) they straight up call their lines “queues.” Queues are First-In-First-Out (FIFO) or in a more simpler way, the earlier you get into the queue, the earlier you can get out.

For example, suppose you are lining up at the cafeteria. Assuming you follow the rules, you get in at the back of the queue and the people who were in front of you get to order their food before you.

Stacks

Stacks are just like a stack of plates. Stacks are Last-In-First-Out, meaning that the ones that enter last are the first to come out.

For example, suppose we have a stack of plates. When you add a plate to the stack, you put it on top. When you take a plate from your stack, you take from the top.

Deques

Deques carry the functionality of both a stack AND a queue. That is, they can append to the front and back. As well, they can pop from the front and back. This means that you can just use a deque instead of a queue/stack if you feel like it.

Main operations using deques:

```
from collections import deque #imports deque from collections

dq = deque() #creates a new deque

dq.append(5) #appends the number 5 to the back of the deque

dq.appendleft(19) #appends the number 19 to the front of the deque

dq.pop() #returns the element at the back of the deque and removes
it from the deque

dq.popleft() #returns the element at the front of the deque and
removes it from the deque

len(dq) #returns the number of elements in the deque
```

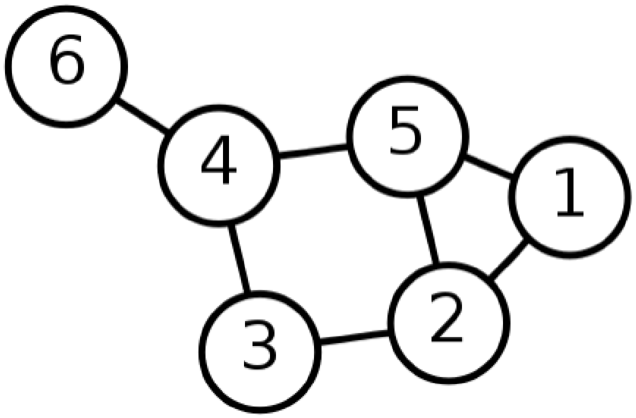
DMOJ Problems:

https://dmoj.ca/problem/stack1
https://dmoj.ca/problem/mccc3j4

Solutions can be found here

Lesson 8 - Graph Theory

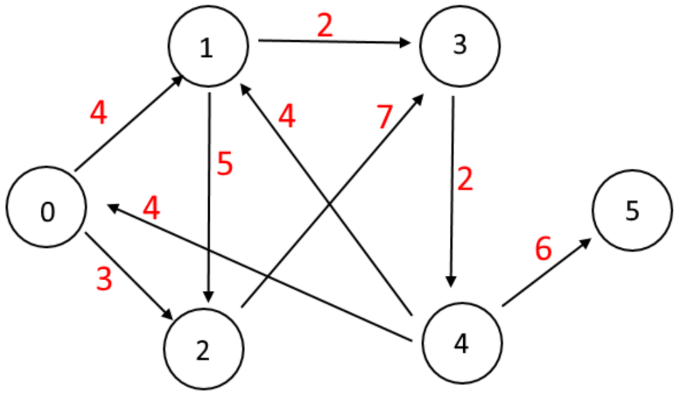
A graph is a data-type that is used to express relationships between certain objects. Typically, we will represent a graph in terms of vertices and edges.



We sometimes call the vertices of a graph **'nodes'**. In the above example, the nodes are '1', '2', '3', '4', '5', and '6.' The lines that connect nodes are called **edges** or links.

We refer to graphs as **directed** if you can only go in a specific direction on each edge, and we call them **undirected** if you can travel both ways on any edge in the graph. Typically, directed graphs will have arrows pointing which direction the edge goes. That is: 1->2 means from 1, you can go to 2 but NOT the other way around.

We can also assign **weights** to edges. Thinking of roads on a map, we know that some roads will take longer to traverse than others because of many factors such as their length, traffic, etc. We thus may sometimes assign weights to all the edges in a graph. Below is an example of a directed and weighted graph.



Adjacency List

An adjacency list is usually a 2-D list with N elements. Each element stores all of the vertices that are connected to the i-th vertex (there are N vertices total.)

The adjacency list for the above graph is:

```
adj = [
    [(1, 4), (2, 3)],
    [(2, 5), (3, 2)],
    [(3, 7)],
    [(4, 2)],
    [(0, 4), (1, 4), (5, 6)]
]
```

Each tuple is of the form (nextNode, weight), meaning there is an edge going to that nextNode with a weight.

Edge List

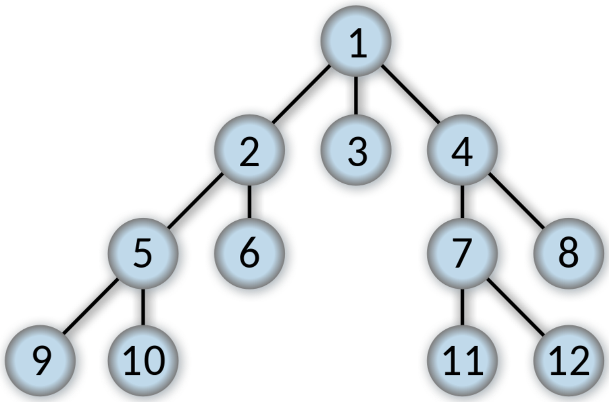
Edge Lists are just as their name implies, they are a list of edges. For example:

```
edges = [
    [1, 2],
    [5, 4],
    [0, 3]
]
```

This means there is an edge from 1->2, 5->4, and 0->3. This is commonly what is given in the input of a graph theory problem.

Breadth-First Search (BFS)

In BFS, nodes are searched one layer at a time. The graph below numbers the order the nodes will be searched. This means, if we start from any vertex on a graph, it will only search the vertices that are closest to the starting vertex first, then it'll only go deeper into the graph once that's completed. We will do this with a queue.



Main applications of BFS include checking if two nodes are connected and the shortest distance between two nodes in an unweighted graph.

Code to check the distance between two nodes:

```
from collections import deque
'''
Using a deque:
q.append(n) --> appends n to the back of the deque
q.appendleft(n) --> appends n to the front of the deque
q.pop() --> removes and returns the element at the back of the deque
q.popleft() --> removes and returns the element at the front of the
deque
len(q) --> returns the number of elements in the deque
'''


graph = [ [1,2],
          [0,2,3],
          [0,1,4,5],
          [1,4],
          [2,3,5,8],
          [2,4,6,7],
          [5,7,8],
          [5,6],
          [4,6,7]
        ]

# setting a visited array to store whether a node has been visited or
not
visited = [False for i in range(len(graph))]

start = 0
end = 8

q = deque()
q.append((start,0)) # We append tuples to the queue
                    # (node number, distance traveled until node)

while len(q) > 0: #keep looping while there are nodes in the deque

    c = q.popleft() # Taking the first element in the queue and
removing it

    node = c[0]
    distance = c[1]

    if node == end: #we reached the node
        print(distance)
        break

    # We add all of the neighboring nodes to the queue
    # provided they are not already visited
    for nxt in graph[node]:
        if not visited[nxt]:
           visited[nxt] = True
            # add 1 to distance
           q.append((nxt, distance+1))
```

Code for the shortest distance between two nodes in general:

```
from collections import deque

numHouses, numRoads, start, dest = map(int, input().split())

visited = [False for i in range(numHouses+1)]
adjacencyList = [[] for i in range(numHouses+1)]

dq = deque()

for i in range(numRoads):
    firstHouse, secondHouse = map(int, input().split())
    adjacencyList[firstHouse].append(secondHouse)
    adjacencyList[secondHouse].append(firstHouse)

dq.append((start, 0))

while len(dq) > 0:
    currentNode, curTime = dq.pop()

    if currentNode == dest:
        print(curTime)
        break
    if visited[currentNode]:
        continue

    visited[currentNode] = True
    for possibleNode in adjacencyList[currentNode]:
        if not visited[possibleNode]:
            dq.append((possibleNode, curTime+1))
```

DMOJ Problems

https://dmoj.ca/problem/vmss7wc16c3p2
https://dmoj.ca/problem/ccc09s3
Template 7-10p BFS problems

Solutions can be found here

Lesson 9 - Recursion

“To understand recursion, you must first understand recursion.”

We’ll start off with a simple example. Compute n!

The notation n! Means $n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1.$

For example,

$$\begin{gathered} 1! = 1 \\ 2! = 2 \cdot 1 = 2 \\ 3! = 3 \cdot 2 \cdot 1 = 6 \\ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \\ 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120. \end{gathered}$$

By definition 0! = 1.

We could run a for loop to compute n!

```
nFactorial = 1

for i in range(1, n+1):
     nFactorial *= i

print(n)
```

However, we can also think of computing this backwards. For a given n, we know that we have to multiply by n-1, then n-2, then n-3, ...

Let’s implement this without using a for loop.

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
```

Recursive Sequence

A recursive sequence is one in which the n^th term is defined by the previous terms in the sequence. One common example of a recursive sequence is the Fibonnaci sequence.

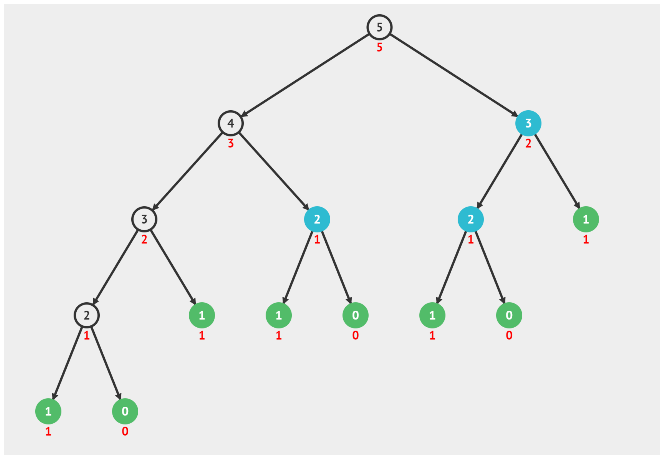
The first two numbers in the Fibonacci sequence are defined as F_0 = 0 and F_1 = 1. For all n > 1, the terms in the Fibonacci sequence are defined as $F_n = F_{n-1} + F_{n-2}$. This means that:

$$F_2 = F_1 + F_0 = 0 + 1 = 1$$
$$F_3 = F_2 + F_1 = 1 + 1 = 2$$
$$F_4 = F_3 + F_2 = 2 + 1 = 3$$

We can similarly compute more terms in the Fibonacci sequence. The main point here is that we have a recurrence relation. There is no obvious general term here so to find the nth term, we rely on knowing the previous terms. In other words, this is a "function" that calls upon itself to compute some term.

For example, say we wanted to find the 6th term of this sequence. We can work backwards with the definition of the Fibonacci sequence.

We know that $F_5 = F_4 + F_3$. We can then further break F_4 and F_3 into $F_4 = F_3 + F_2$ and $F_3 = F_2 + F_1$. If we continue breaking these parts, we will eventually have everything in terms of F_0 and F_1 which we are both 1. This is demonstrated below.



We can implement this using a function.

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n-1) + fib(n-2)

print(fib(5))
print(fib(10))

-----
Output: 5
Output: 55
```

Whenever we have a recursive function. We have to be careful of two things.

Base Case

There must always be a base case at which the function will not call itself. If a base case did not exist, then the function would keep calling itself indefinitely. In the Fibonacci Sequence, we have two base cases: when n = 0 and n = 1.

Each Recursive Call gets Closer to the Base Case

Just like a while loop, your recursion needs to end at some point.

When you implement a recursive function, you may get the following error.

```
RecursionError: maximum recursion depth exceeded
```

This means that your function keeps calling on itself and doesn't reach a base case. In python, there is a limit to how many times a function can call itself recursively. You have to make sure that with each recursive call, your function will approach the base case so that it does not run forever.

If you think you have set up your recursion properly but you still get RecursionError, you can increase the recursion depth of python. By default, the value is set to 10^3.

```
import sys
sys.setrecursionlimit(10**9)
```

DMOJ Problems

https://dmoj.ca/problem/ccc96s3
https://dmoj.ca/problem/dwite09c1p2
https://dmoj.ca/problem/occ19s2

Solutions can be found here

Lesson 10 - Dictionaries

Dictionaries

Dictionaries store values in the from {key : value}. That is, for every key, there is ONE value. That is there can NOT be two values for one key. However, you may have a key pair with a list or even another dictionary!

There are two main ways to create a dictionary:

```
dictionary_name = {}
dictionary_name = dict()
```

This is similar to the two main ways to create a list:

```
list_name = []
list_name = list()
```

Assigning values is done just like assigning a variable.

```
marks["Alan"] = 86
```

To retrieve the value at a key, it's just like indexing a list, except instead of list index, you put the key.

```
print(marks["Alan"]) #prints the value with key "Alan" in the
marks dictionary
```

Why use a dictionary? Dictionaries are used for similar reasons as a list, except can store keys which are not just numbers. Also, to access dictionary[2000000000], there does not need to be 2000000001 elements like a list. This way, you can store larger and more different data types for keys.

Looping through dictionary keys is just like looping through a list

```
marks = {
    "Alan": 86,
    "Amanbir": 100,
    "Evan": 130
}

for key in marks:
    print(key)
```

In the case that you want to delete an element from a dictionary, you can use the del keyword. Del can also delete the entire dictionary if you are worried about using too much memory. To clear the dictionary, use clear()

```
del marks["Alan"] #deletes the "Alan" key
del marks #deletes the entire dictionary
marks.clear() #clears all keys and values in the dictionary
```

Dictionaries inside of dictionaries are like lists inside of lists. Each time you use a key, you go one layer into the dictionary until you reach the last value.

```
school = {
    "teachers" : {
        "names": ["Alan", "Evan", "Amanbir", "Rayton"]
    },
    "students": {
        "average":{
            "Alan" : 86,
            "Evan": 130
        }
    }
}
```

DMOJ Problems

https://dmoj.ca/problem/alphabetscore
https://dmoj.ca/problem/sac22cc5jp3
https://dmoj.ca/problem/tle16c6s1
https://dmoj.ca/problem/ccc22s2

Solutions can be found here

Lesson 11 - Classes

In the past, we have used many built-in types such as int, string, and float. Now, let's make our own type. We could call it anything we want just like variables. This is referred to as making a class! In doing so, we introduce the idea of Object Oriented Programming. As an analogy, your class is a cookie cutter and your instance of the class (object) is the cookie. Classes serve as a good template for your objects which can avoid repeatability in your code. In real life, you might not want to objectify people, but as an example, let's objectify Alan.

Alan:

Fields (properties of Alan):

- Age 16
- Wears glasses is True
- Asian is True
- Black hair is True
- Ate 11.4 chocolate bars in the last week

Methods/Functions (things Alan can do):

- Walk()
- Run()
- Talk()

In Python, if we were able to make a general object for a Person with these fields, we could have many different Persons with the same information. We would want to do this if we have multiple Persons in our program. In general, a Person class would look something like this:

Person:

Fields (properties of Person):

- Age <int>
- wearsGlasses <boolean>
- race <string>

- hairColor <string>

Methods/Functions (things Person can do):

- Walk()
- Run()
- Talk()

To be clear, this is pseudo-code, not real Python code.

For example, in Python:

```
class Person():
    def __init__(self, name, yearsOld, glasses, race, hair):
#initializes values of the Person, this method will happen
whenever you make an instance of this class
        self.age = yearsOld
        self.name = name
        self.wearsGlasses = glasses
        self.race = race
        self.hairColor = hair

    def talk(self):
        print("Hi, my name is", self.name)
        print("I am", self.age, "years old")

        if self.wearsGlasses:
            print("I wear glasses")
        else:
            print("I do not wear glasses")

        print("I am", self.race, "and I have", self.hairColor,
"hair")

Alan = Person("Alan", 16, True, "Asian", "Black")
Alan.talk()
```

Note: self refers to the current object being worked with. For example, while we’re in Alan, and we want to output Alan’s age, we use self.age. In other words, methods that a Person has might want to pass the Person in as a variable.

Overall, classes are not mandatory to use most of the time, but they can be used to make code cleaner. You may not use them in competitive programming much yet, but it’s useful to know for the future!

Editorials and Solutions

Below, there are solutions to the problems discussed in this document. Some have a written explanation with them, but some do not.

Lesson 1

Hello, World!

The problem asks to print out the message Hello, World! In order to do so, you should use the print keyword. The code should look like this:

```
print("Hello, World!")
```

CCC ‘13 J1 - Next in Line:

The problem gives you the youngest and middle child’s ages and asks you to determine the oldest child’s age, given that the difference between the oldest child’s age and the middle child’s age is equal to the difference between the middle child’s age and the youngest child’s age.

We can benefit from using the Samples to our advantage.

Sample 1:
youngestChildAge = 12
middleChildAge = 15
=> oldestChildAge = 18

Sample 2:
youngestChildAge = 10
middleChildAge = 10
=> oldestChildAge = 10

In Sample 1, it can be noted that 15-12 = 3, and as well, 18-15 = 3.
In Sample 2, it can be noted that 10-10 = 0 and as well, 10-10 = 0.

We can observe that if we know the difference, we can just add it to the middle child's age to get the oldest child's age.

The first step of the problem is to take in input.
Then, we want to convert the input from string (input() is string by default) to integer so we can add and subtract numbers.
Next, we want to find the difference between the middle child's age and the youngest child's age.
Lastly, add the difference to the middle child's age and then print it out since that is the oldest child's age.

```
youngestChildAge = input()
middleChildAge = input()
youngestChildAge = int(youngestChildAge)
middleChildAge = int(middleChildAge)

difference = middleChildAge - youngestChildAge

oldestChildAge = middleChildAge + difference

print(oldestChildAge)
```

CCC '21 J1 - Cupcake Party

After taking input and converting the input to integers, the first step is to determine the total number of cupcakes that you have. Each regular box contains 8 cupcakes and each small box contains 3 cupcakes. Therefore, the total_number_of_cupcakes = (8*regular_box) + (3*small_box) where regular_box and small_box are the number of regular and small boxes, respectively. If each student gets one cupcake and there are 28 students, the leftovers = total_number_of_cupcakes - 28.

```
regular_box = input()
regular_box = int(regular_box)
small_box = input()
small_box = int(small_box)

total_number_of_cupcakes = (8*regular_box) + (3*small_box)
leftovers = total_number_of_cupcakes - 28
print(leftovers)
```

Lesson 2

GFSSOC ‘15 Winter J1 - Festive Fardin

The problem asks you to determine if both the shirt and pants colors are one of red, green, or white. If so, output Jingle Bells. Otherwise, output Boring...

This can be done using an if statement. To check if shirt is red, green, or white AND pants is red, green, or white:

```
#first take input
shirt = input()
pants = input()

if (shirt == red or shirt == green or shirt == white) and (pants
== red or pants== green or pants== white):
     print("Jingle Bells")
else:
     print("Boring...")
```

Notice the use of the brackets. Python will check the conditions in the brackets first!

CCC ‘08 J1 - Body Mass Index

Given the weight and height (in that order), first determine the BMI. We want to use floating point numbers here because there could be decimals.

Using the formula given in the problem statement:

```
#take input as floats
weight = float(input())
height = float(input())

bmi = weight / (height*height)
```

```
#With if statements, we can check which category the person
belongs to based on the values in the table
if bmi > 25:
     print("Overweight")
elif bmi < 18.5:
     print("Underweight")
else:
     print("Normal weight")
```

CCC ‘20 J1 - Dog Treats

First, we have to determine the dog’s happiness value. From the problem statement, that is 1*S + 2*M + 3*L. Then, we must check if it’s greater than or equal 10 or not.

```
#take input
S = int(input())
M = int(input())
L = int(input())
happiness = (1*S) + (2*M) + (3*L)

if happiness >= 10:
     print("happy")
else:
     print("sad")
```

CCC ‘17 J1 - Quadrant Selection

Notice that:

The answer is Quadrant 1 if x > 0 and y > 0
The answer is Quadrant 2 if x < 0 and y > 0
The answer is Quadrant 3 if x < 0 and y < 0
The answer is Quadrant 4 if x > 0 and y < 0

We can check this with 4 if statements. Make sure to take input as ints!

```
x = int(input())
y = int(input())

if x > 0 and y > 0:
     print("1")
if x < 0 and y > 0:
     print("2")
if x < 0 and y < 0:
     print("3")
if x > 0 and y < 0:
     print("4")
```

CCC ‘21 J1 - Boiling Water

The first task is to find the atmospheric pressure, P, using the formula given:

P = 5*B - 400

Once that is solved, we should determine if it is above, below, or on sea level. From the sample, we can see that when P < 100, you are above (output 1); when P > 100, you are below (output -1); when P == 100 (output 0), you are at sea level.

```
B = int(input()) #B is given in the input
P = 5*B - 400
print(P)

if P < 100:
     print(1)
if P == 100:
     print(0)
if P > 100:
     print(-1)
```

CCC ‘07 J1 - Who is in the middle?

Let a, b, c, represent the first, second, and third child’s ages respectively. There are three cases:

a is in the middle: b <= a <= c or c <= a <= b
b is in the middle: a <= b <= c or c <= b <= a
c is in the middle: a <= c <= b or b <= c <= a

Now, we just have to check these three cases

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

if b <= a <= c or c <= a <= b:
    print(a)
elif a <= b <= c or c <= b <= a:
    print(b)
else:
    print(c)
```

CCC ‘18 J1 - Telemarketer or not?

We should go through all the conditions to check if the number belongs to a telemarketer.

If the first digit is not 8 or 9, answer.
If the last digit is not 8 or 9, answer.
If the second and third digits are not equal, answer.
Otherwise, all the statements are true and we do NOT want to answer, so we ignore

We can either combine this into one statement, or do it separately

```
#one big statement
digit1 = int(input())
digit2 = int(input())
digit3 = int(input())
digit4 = int(input())

if (digit1 != 8 and digit1 != 9) or (digit4 != 8 and digit4 != 9)
or (digit2 != digit3):
     print("answer")
else:
     print("ignore")
```

#multiple statements

```
digit1 = int(input())
digit2 = int(input())
digit3 = int(input())
digit4 = int(input())

if digit1 == 8 or digit1 == 9:
     if digit4 == 8 or digit4 == 9:
          if digit2 == digit3:
               print("ignore")
          else:
               print("answer")
     else:
          print("answer")
else:
     print("answer")
```

Lesson 3

Valentine's Day '19 J1 - Rainbow Rating

Use a for loop to loop through each case since we know exactly how many cases there are going to be. Then, use if statements to determine the correct title.

```python
N = int(input())

for i in range(N):
    rating = input()
    rating = int(rating)
    if rating < 1000:
           print("Newbie")
    elif 1000 <= rating and rating <= 1199:
        print("Amateur")
    elif 1200 <= rating and rating <= 1499:
        print("Expert")
    elif 1500 <= rating and rating <= 1799:
        print("Candidate Master")
    elif 1800 <= rating and rating <= 2199:
        print("Master")
    elif 2200 <= rating and rating <= 2999:
        print("Grandmaster")
    elif 3000 <= rating and rating <= 3999:
        print("Target")
    else:
        print("Rainbow Master")
```

ACM U of T Tryouts C0 A - Max Flow

Use a for loop to loop through each case since we know exactly how many cases there are going to be. For each case, loop through all the flows while maintaining the highest flow.

```
times = int(input())
GARBAGE = -4000000000000 #COMPLETELY out of the range of values

for i in range(times): #run "times" times
    #will process each scenario

    numFlows = int(input())
    highestSoFar = GARBAGE

    for x in range(numFlows): #run numFlows times
        flowVal = int(input())

        highestSoFar = max(highestSoFar, flowVal)

    print(highestSoFar)
```

CCC ‘21 J2 - Silent Auction

Use a for loop to loop through each person since we know exactly how many people there are going to be. Keep track of the highest bid and who bid that number.

```
numPeople = int(input())
INF = int(10**9)
#-INF is a very small number

highestSoFar = -INF
highestBidder = ""
```

```
for i in range(numPeople):
    name = input()
    money = int(input())

    if money > highestSoFar:
        highestBidder = name
        highestSoFar = money

print(highestBidder)
```

CCC '20 J2 - Epidemiology

Use a while loop to continue while the total number of people infected is less than the limit, given in the input. After one day, each person who was infected infects someone else and does NOT infect any more people, effectively becoming uninfected. Thus, we can keep track of the total number of people infected and the number of people that can infect to solve the problem.

```
limit = int(input())
numOnDay0 = int(input())
rate = int(input())

totalInfected = numOnDay0
curInfected = totalInfected
curDay = 0

while totalInfected <= limit:
    totalInfected += curInfected*rate
    curInfected = curInfected*rate
    curDay += 1

print(curDay)
```

VM7WC '15 #2 Bronze - Recruits!

While taking input, keep track of the previous, current, and next person's height. Keep a counter that increases when previous <= 41, current <= 41, and next <= 41.

```
n = int(input())

prev = -1
cur = int(input())
ans = 0

for i in range(n-1):

    nxt = int(input())

    if prev <= 41 and cur <= 41 and nxt <= 41:
        ans += 1

    prev = cur
    cur = nxt

if cur <= 41 and prev <= 41:
    ans += 1
print(ans)
```

Lesson 4

Sorting

Store numbers in a list. Then use python’s built-in sort.

```
listLength = int(input())
numbers = []

for i in range(listLength):
    numbers.append(int(input()))

numbers.sort()

for i in range(listLength):
    print(numbers[i])
```

List Minimum (Hard)

Same as Sorting. Don’t simulate everything the statement says to. What matters here is the output.

```
listLength = int(input())
numbers = []

for i in range(listLength):
    numbers.append(int(input()))

numbers.sort()

for i in range(listLength):
    print(numbers[i])
```

CCC ‘11 S2 - Multiple Choice

Keep track of student’s and teacher’s responses in lists. Compare and keep track of if the student’s answer matches with the teacher’s.

```python
n = int(input())

yourAnswers = []
correctAnswers = []

for i in range(n):
    ans = input()
    yourAnswers.append(ans)

for i in range(n):
    refAns = input()
    correctAnswers.append(refAns)

points = 0

for i in range(n):
    if yourAnswers[i] == correctAnswers[i]:
        points += 1

print(points)
```

CCC ‘97 S1 - Sentences

Keep track of subjects, verbs, and objects in three separate lists. Loop through them and output the sentence. Since they are given in alphabetical order in the input, this is sufficient. Be careful to reset the lists for each data set.

```python
dataSets = int(input())
```

```
for i in range(dataSets):
    numSubjects = int(input())
    numVerbs = int(input())
    numObjects = int(input())

    subjects = []
    verbs = []
    objects = []

    for j in range(numSubjects):
        subjects.append(input())

    for j in range(numVerbs):
        verbs.append(input())

    for j in range(numObjects):
        objects.append(input())

    for subject in subjects:
        for verb in verbs:
            for obj in objects:
                print(subject, verb, obj+".")
```

CCC ‘03 S1 - Snakes and Ladders

Simulate the instructions given in the problem statement. Lists can be used to keep track of where the snakes/ladders are, but since there are so few of them, it isn’t even necessary.

```
squarenum=1
snake=[19, 48,77]
snakesquare=[54, 90, 99]
ladder=[34, 64, 86]
laddersquare=[9, 40, 67]
```

```
while 1<=squarenum and squarenum<=100:
  step=int(input())
  squarenum=step+squarenum

  if step==0:
    print("You Quit!")
    break

  elif squarenum==100:
    print("You are now on square 100")
    print("You Win!")
    break

  elif squarenum>100:
    squarenum=squarenum-step
    print("You are now on square"+" "+str(squarenum))

  elif squarenum not in snakesquare and squarenum not in
laddersquare:
    print("You are now on square"+" "+str(squarenum))

  elif squarenum in snakesquare:
    slide=int(snakesquare.index(squarenum))
    squarenum=snake[slide]
    print("You are now on square"+" "+str(squarenum))

  elif squarenum in laddersquare:
    climb=int(laddersquare.index(squarenum))
    squarenum=ladder[climb]
    print("You are now on square"+" "+str(squarenum))
```

GFSS Christmas Challenge P3 - Candy Distribution

It can be proven that it is optimal to give no more than c_i candies to any reindeer. Additionally, it can be proven that it is optimal to give candies to the reindeer who wants the least amount of candy each time. This problem is then just like List Minimum (Hard).

```
numReindeer, numCandies = input().split()
numReindeer = int(numReindeer)
numCandies = int(numCandies)
satisfied = 0
candiesWanted = []

for i in range(numReindeer):
    candiesWanted.append(int(input()))

candiesWanted.sort()

for i in range(numReindeer):
    if numCandies >= candiesWanted[i]:
        numCandies -= candiesWanted[i]
        satisfied += 1

print(satisfied)
```

Mispelling

There may be spaces in the word they give, so input().split() will not work how we want it to. Instead, we can keep track of where the first space is. The indices up to the space is the index we want to remove and the string from space+1 until the end is the word we are outputting.

```python
n = int(input())

for i in range(n):
    line = input()
    space= line.index(' ')

    index = int(line[0:space])-1
    word = line[space+1: ]

    print(i+1, word[0:index] + word[index+1: ])
```

Lesson 5

3n + 1

Make a function to compute the next number of the sequence. Loop while n != 1.

```python
def nextNum(n): #returns the next number in the sequence
    if n%2 == 0: #if n is even
        return n//2
    if n%2 == 1: #if n is odd
        return 3*n +1

def solve():
    N = int(input())
    times =0

    while N != 1:
        N = nextNum(N)
        times += 1

    print(times)

solve()
```

Back From Summer ‘17 P1: Pithy Pastimes

Loop through hobbies. Increase a counter if the word is valid.

```python
def isValid(word): #returns True if the word is valid, otherwise
returns False
    if len(word) > 10: #invalid word
        return False
    return True #valid word

numHobbies = int(input())
```

```
hobbies = input().split()

count = 0
for hobby in hobbies:
    if isValid(hobby):
        count += 1

print(count)
```

Next Prime

Create a function to check if a number is prime. Loop until you reach a prime number.

```
def isPrime(n):
    if n <= 1:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i*i <= n:
        if n % i == 0:
            return False
        i += 2

    return True

n = int(input())
while True:
    if isPrime(n):
        print(n)
        break
    else:
        n += 1
```

Lesson 6

Big Chess

We can represent chess boards as a 2D-list! Given the width and height of the board, a board filled with all 1s would be:

```
board = [[1 for i in range(width)] for k in range(height)]
```

Expected output of a 4x4 board (the bold numbers are the board, the non-bold are the row/column number):

	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

Notice which ones are 0s and see if there's a pattern. Some squares filled with 0: (0,0), (0, 2), (1, 1), (1, 3), (2, 0), (2, 2), (3, 1), (3, 3). Something we can notice is that when both the coordinates are odd OR both the coordinates are even, it is a 0. Therefore, we can run a for loop and do:

```
if innerList%2 == idx%2:
    board[innerList][idx] = 0
```

Implementation:

```
width, height = map(int, input().split())
board = [[0 for i in range(width)] for k in range(height)]

for innerList in range(height):
    for idx in range(width):
        if innerList % 2 == idx % 2:
            board[innerList][idx] = 1

for i in range(height):
    print(*board[i], sep = "")
```

Note:
print(*lis) will output lis with the elements separated by space. To change the separator add sep = “whatever you want the separator to be”

Queens can't attack me!

For each queen, we want to check all directions the queen can go and mark it somehow. We can create a 2D-list of booleans called board, where board[i][k] == False iff the square is not reachable by any queen. Pseudocode:

```
for each queen:
    mark horizontal
    mark vertical
    mark diagonals
```

```
n, m = map(int, input().split())
board = [[False for i in range(n)] for k in range(n)]

def onBoard(x, y, n):
    if 0 <= x < n and 0 <= y < n:
        return True
    return False

def markHorizontal(innerList, idx, n):

def markVertical(innerList, idx, n):
```

```
def movingDownAndLeft(innerList, idx, n):

def movingUpAndRight(innerList, idx, n):

def movingUpAndLeft(innerList, idx, n):

def movingDownAndRight(innerList, idx, n):

def markDiagonals(innerList, idx, n):
    movingDownAndLeft(innerList, idx, n)
    movingDownAndRight(innerList, idx, n)
    movingUpAndLeft(innerList, idx, n)
    movingUpAndRight(innerList, idx, n)

for i in range(m):
    x, y = map(int, input().split())
    #get coordinates to 0-indexing
    x -= 1
    y -= 1

    markHorizontal(x, y, n)
    markVertical(x, y, n)
    markDiagonals(x, y, n)

ans = 0
for i in range(n):
    for k in range(n):
        if board[i][k] == False:
            ans += 1

print(ans)
```

Lesson 7

The DMOJ Driveway

```
from collections import deque

t, m = map(int, input().split())

dq = deque()

for i in range(t):
    name, op = input().split()

    if op == "in":
        dq.append(name)
    else:
        carLeaving = dq.pop()
        if carLeaving != name:
            dq.append(carLeaving)
        if m == 0:
            continue

        carLeaving = dq.popleft()

        if carLeaving != name:
            dq.appendleft(carLeaving)
        else:
            m -= 1

for name in dq:
    print(name)
```

Mock CCC ‘22 Contest 1 J4 - Snowball Fight

```python
from collections import deque

numPeople, rounds = map(int, input().split())
firstThrow = [-1] + [int(i) for i in input().split()]
hitList = [-1] + [deque() for i in range(numPeople)]
#hitList[i] = stores the hitlist of the people the ith person
wants to throw a snowball at

lastThrown = [-1 for i in range(numPeople+1)]

for person in range(1, numPeople+1):
    hitList[person].appendleft(firstThrow[person])

for i in range(rounds):
    threwTo = [-1 for i in range(numPeople+1)]

    for person in range(1, numPeople+1):
        if len(hitList[person]) == 0:
            continue

        personGettingHit = hitList[person].popleft()
        lastThrown[person] = personGettingHit
        threwTo[person] = personGettingHit

    for person in range(1, numPeople+1):
        if threwTo[person] == -1:
            continue

        hitList[threwTo[person]].append(person)

print(*lastThrown[1: ])
```

Lesson 8

VM7WC '16 #3 Silver - Can Shahir even get there?!

A BFS approach is shown below

```python
from collections import deque

numHouses, numRoads, start, dest = map(int, input().split())

visited = [False for i in range(numHouses+1)]
adjacencyList = [[] for i in range(numHouses+1)]

for i in range(numRoads):
    firstHouse, secondHouse = map(int, input().split())
    adjacencyList[firstHouse].append(secondHouse)
    adjacencyList[secondHouse].append(firstHouse)

q = deque()
q.append(start)

while len(q) > 0:
    curNode = q.popleft()
    visited[curNode] = True
    possibleNodes = adjacencyList[curNode]

    for nxtNode in possibleNodes:
        if not visited[nxtNode]:
            q.append(nxtNode)

if visited[dest]:
    print("GO SHAHIR!")
else:
    print("NO SHAHIR!")
```

CCC '09 S3 - Degrees Of Separation

```python
from collections import deque

numPeople = 51

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

def separation(start, dest):
    q = deque()
    q.append(start)
    inf = int(10**9)+7
    distance = [inf for i in range(51)]
    distance[start] = 0

    while len(q) > 0:
        curPerson = q.popleft()

        for nxtPerson in friendList[curPerson]:
            if distance[nxtPerson] > distance[curPerson] + 1:
                distance[nxtPerson] = distance[curPerson] + 1
                q.append(nxtPerson)

    if distance[dest] == inf:
        return "Not connected"
    return distance[dest]

def totalFriends(start):
    q = deque()
    q.append(start)
    inf = int(10**9)+7
    distance = [inf for i in range(51)]
    distance[start] = 0
    while len(q) > 0:
        curPerson = q.popleft()

        for nxtPerson in friendList[curPerson]:
            if distance[nxtPerson] > distance[curPerson] + 1:
                distance[nxtPerson] = distance[curPerson] + 1
                q.append(nxtPerson)

    ans = 0
    for dist in distance:
        if dist < 3:
            ans += 1
    return ans
lis = []
while True:
    operation = input()
    if operation == "q":
        break
    if operation == "i":
        x = int(input())
        y = int(input())
        if y not in friendList[x]:
            friendList[x].append(y)
            friendList[y].append(x)

    if operation == "d":
        x = int(input())
        y = int(input())
        friendList[x].remove(y)
        friendList[y].remove(x)

    if operation == "n":
        x = int(input())
        lis.append(len(friendList[x]))

    if operation == "f":
        x = int(input())
        lis.append(totalFriends(x) - len(friendList[x])-1)

    if operation == "s":
        x = int(input())
        y = int(input())
        lis.append(separation(x, y))

for i in lis:
    print(i)
```

Lesson 9

CCC '96 S3 - Pattern Generator

```
def bits(n, cur, limit): #number of bits, string of the current
configuration, limit to the number of 1s
    if cur.count("1") > limit: #too many ones
        return
    if n == 0: #BASE CASE
        if cur.count("1") == limit:
            print(cur)
        return

    #n-1 approaches the base case, n = 0
    bits(n-1, cur+"1", limit)
    bits(n-1, cur+"0", limit)


times = int(input())
for i in range(times):
    print("The bit patterns are")
    n, k = map(int, input().split())

    bits(n, "", k)
    if i != times-1: #doesn't put an extra newline at the end
        print()
```

DWITE '09 R1 #2 - Word Scrambler

```python
def recur(possibleLetters, curWord, expectedLength):
    if len(curWord) == expectedLength: #base case
        print(curWord)
        return

    for i in range(len(possibleLetters)):
        letter = possibleLetters[i]
        recur(possibleLetters[0:i]+possibleLetters[i+1: ],
curWord+letter, expectedLength) #the length of curWord increases,
approaching the base case

for i in range(5):
    word = input()
    recur(word, "", len(word))
```

OCC '19 S2 - Rimuru's Number Game

```python
n = int(input())
ans = 0

def solve(curNum):
    global ans
    if int(curNum+"2") > n: #base case
        return

    if int(curNum+"2") <= n:
        ans += 1
        solve(curNum+"2") #curNum increases, approaching the base
case

    if int(curNum+"3") <= n:
        ans+=1
        solve(curNum+"3") #curNum increases, approaching the base
case

solve("")
print(ans)
```

Lesson 10

TLE '16 Contest 6 (Mock CCC) S1 - Writing the CCC

```python
numTypes = int(input())
types = []
test = {}
#{problemType : [indices of this problem type]}

for i in range(numTypes):
    name = input()
    test[name] = []
    types.append(name)

problems = []
numProblems = int(input())
for i in range(1, numProblems+1):
    problemType = input()
    test[problemType].append(i)
    problems.append(problemType)

for pType in types: #runs at most 10**5 times
    for index in test[pType]: #runs at most 10**5 more times
        print(index)
```

Alphabet Score

There are many ways to solve this problem.

Using ASCII values:

```
string = input()
alphaScore = 0

for let in string: #runs len(string) times
    # print(let)
    alphaScore += ord(let)

print(alphaScore - 96*len(string))
```

Using a dictionary:

```
word = input()
num = 0
val = {"a" : 1,
    "b" : 2,
    "c" : 3,
    "d" : 4,
    "e" : 5,
    "f" : 6,
    "g" : 7,
    "h":8,
    "i":9,
    "j":10,
    "k":11,
    "l":12,
    "m":13,
    "n":14,
    "o":15,
    "p":16,
    "q":17,
    "r":18,
    "s":19,
    "t":20,
    "u":21,
    "v":22,
    "w":23,
    "x":24,
    "y":25,
    "z":26}

for i in range(len(word)):
  num += val[word[i]]

print(num)
```

CCC ‘22 S2 - Good Groups

Dictionary is used to map each person to a group number. It can be easily checked if two people are in the same group by comparing their group numbers.

```python
X = int(input()) #must be in same group
lisX = [input().split() for i in range(X)]

Y = int(input()) #must NOT be in same group
lisY = [input().split() for i in range(Y)]

G = int(input()) #of groups

groupNumber = {}

for curNum in range(G):
    names = input().split()
    for name in names:
        groupNumber[name] = curNum

ans = 0

for name1, name2 in lisX:
    if groupNumber[name1] != groupNumber[name2]:
        ans += 1

for name1, name2 in lisY:
    if groupNumber[name1] == groupNumber[name2]:
        ans += 1

print(ans)
```