# Artificial Intelligence II

Lesson 2 - Graphs and Matrices

First Code Academy

# Today's Plan

| | |
|---|---|
| Introduction | 00 - 5 min |
| Matrices | 10 - 20 min |
| Graphs | 20 - 25 min |
| Quiz | 25 - 30 min |
| Break | 30 - 35 min |
| Project - Shortest path between cities | 35 - 90 min |

# Teach Back

- What did we learn last class?

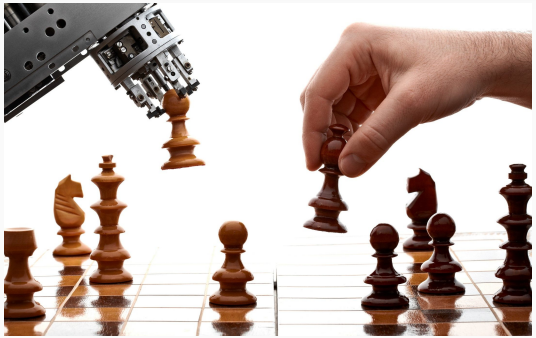
- What did we make last time?

# Teach Back

- _____ is just making the computer smart
- An _____ is a series of steps to accomplish a goal
- _____ refers to how much time and memory resources our algorithm requires
- _____ search does not know anything else about the environment

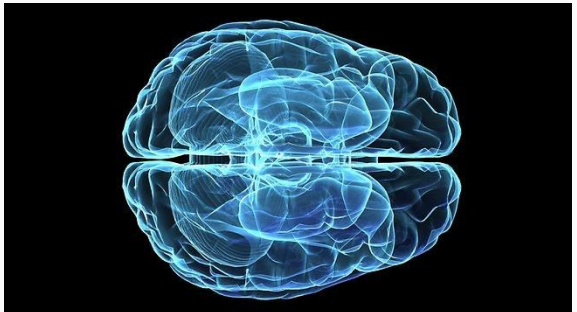
# 3 Waves of AI

Wave 1: Weak (Logic based)



Wave 2: Intermediate (Experience based)

Wave 3: Strong (Hybrid?)



time

# What has AI done recently?

GPT3 is a Natural Language Processing system that can produce very human-like text!

```
Q: What is your favorite animal?
A: My favorite animal is a dog.

Q: Why?
A: Because dogs are loyal and friendly.

Q: What are two reasons that a dog might be in a bad mood?
A: Two reasons that a dog might be in a bad mood are if it is hungry or if it is hot.

Q: How many eyes does a giraffe have?
A: A giraffe has two eyes.

Q: How many legs does a frog have?
A: A frog has four legs.

Q: Are there any animals with three legs?
A: No, there are no animals with three legs.

Q: Why don't animals have three legs?
A: Animals don't have three legs because they would fall over.
```

# What has AI done recently?

A team of AI bots won The International, the biggest tournament in Dota 2

# Key Terms

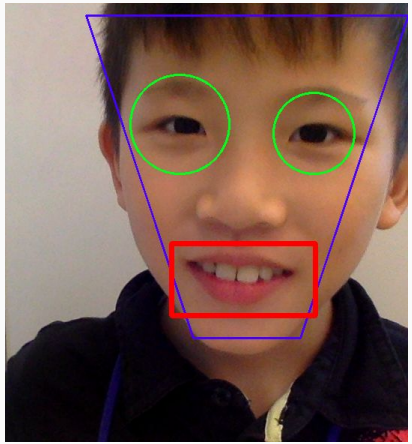
Matrices

Graphs

# Matrices

# Matrices

How does our computer understand images?



Original image

```
[[[12 11 15]
  [ 9 11 12]
  [ 7 12 11]
  ...
  [22 31 21]
  [21 29 18]
  [24 32 21]]

 [[12 13 17]
  [11 12 16]
  [10 14 15]
  ...
```

Computer's understanding

```
[[r g b]
...
[r g b]]
```

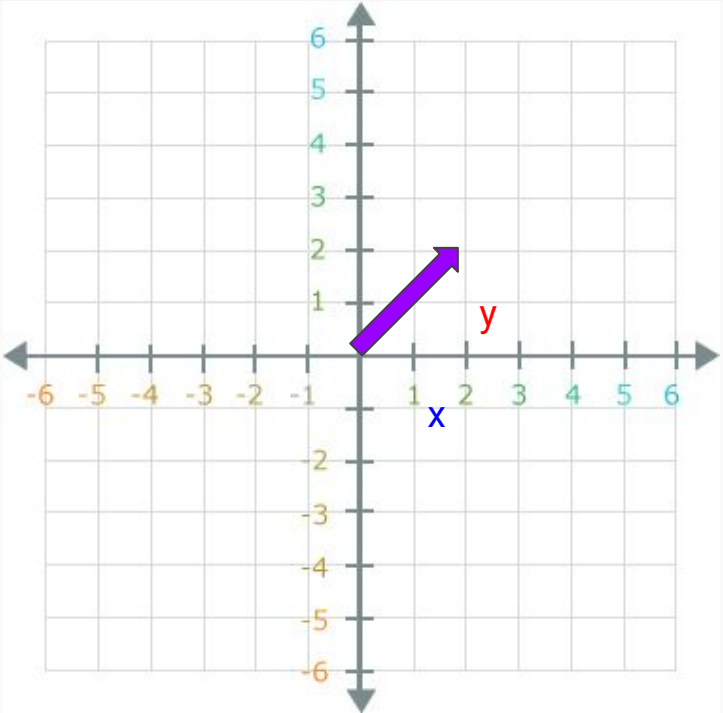
Your computer sees images as a giant matrix of red, green and blue values, one for each pixel

# Matrices

A matrix is made up of **vectors**.

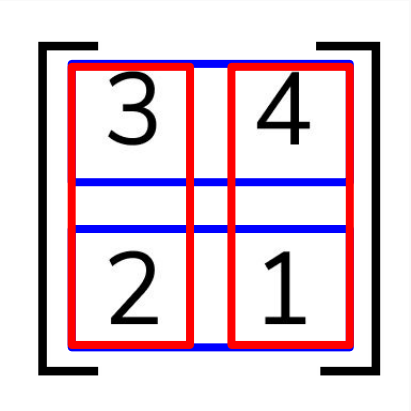
A **vector** is just a position (x, y) on a plane

With just x and y we can reach any point on this plane



# Matrices

We write matrices like this:




2 rows

2 columns

# Matrices

There are two basic operations with matrices:

## Matrix Addition

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 5 \\ 3 & 7 \end{bmatrix} = \begin{bmatrix} 4 & 9 \\ 5 & 8 \end{bmatrix}$$

Matrix 1 Matrix 2 Resultant Matrix

## Matrix Multiplication

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 5 \\ 3 & 7 \end{bmatrix} = \begin{bmatrix} 3+12 & 15+28 \\ 2+3 & 10+7 \end{bmatrix}$$

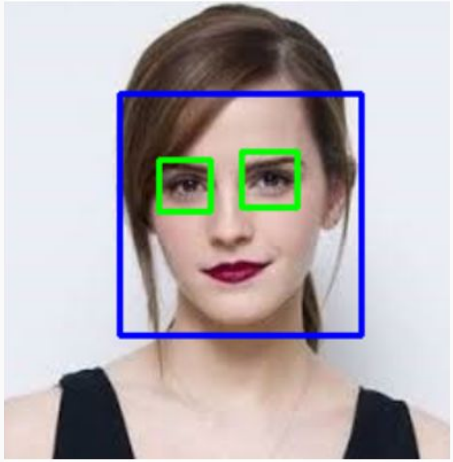
Matrix 1 Matrix 2

$$= \begin{bmatrix} 15 & 43 \\ 5 & 17 \end{bmatrix}$$

Resultant Matrix

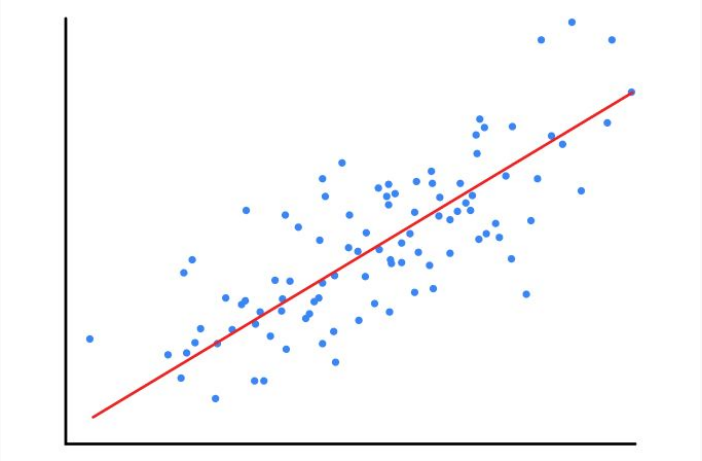
# Matrices in AI

Lots of math! Where could we use it in AI?



Facial Recognition

You could group any related information, really.



Finding patterns in data

# Matrices in Python

In Python,  matrices are just made up of lists!

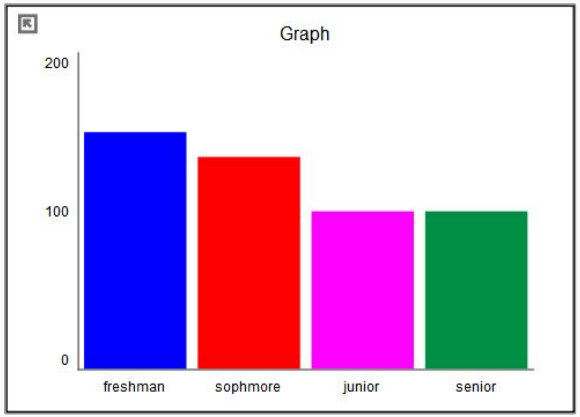
```
oneDimensionalMatrix = [1, 2, 3, 4, 5]
```

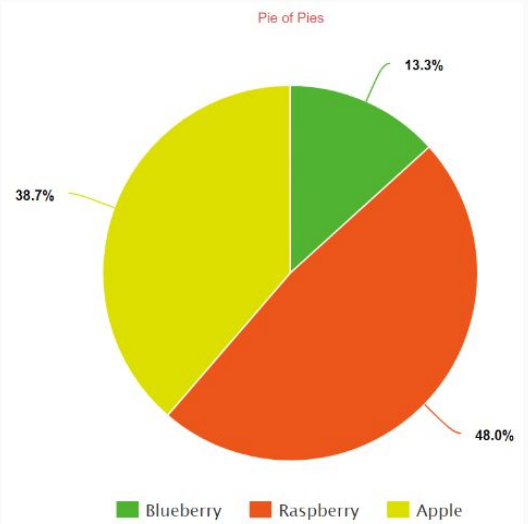
```
twoDimensionalMartix = [[1, 2, 3],
                        [4, 5, 6],
                        [7, 8, 9],
                        [10, 11, 12]]
```

# Graphs

# Graphs

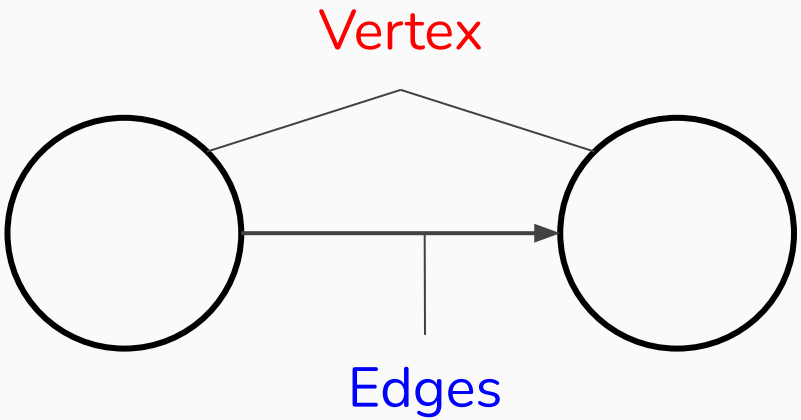
We might have heard of these types of graphs:



# Graphs

In math, graphs look.....quite different!



# Graphs

**Directed graphs** have an arrow pointing the direction in which they are connected

**Undirected graphs** have edges which connect both ways

Undirected Graph

Directed Graph

# Graphs

**Weighted** edges have a cost associated with taking them

**Unweighted** edges have no cost (or they all cost the same)

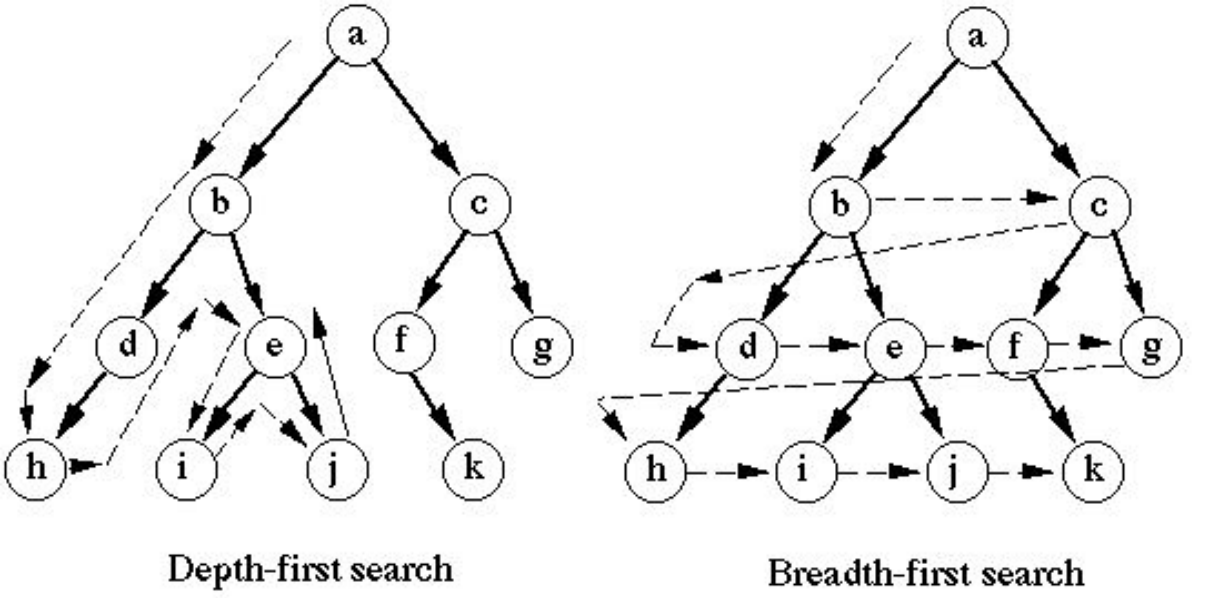
Unweighted


Weighted

# Graphs in searching

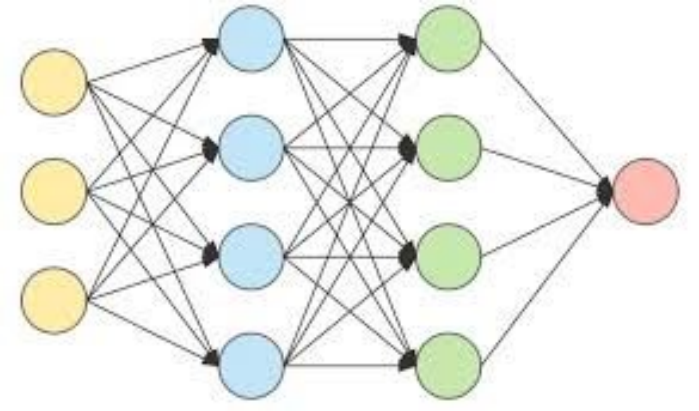
We've already seen graphs before!

Are trees and graphs similar?



# Graphs in AI

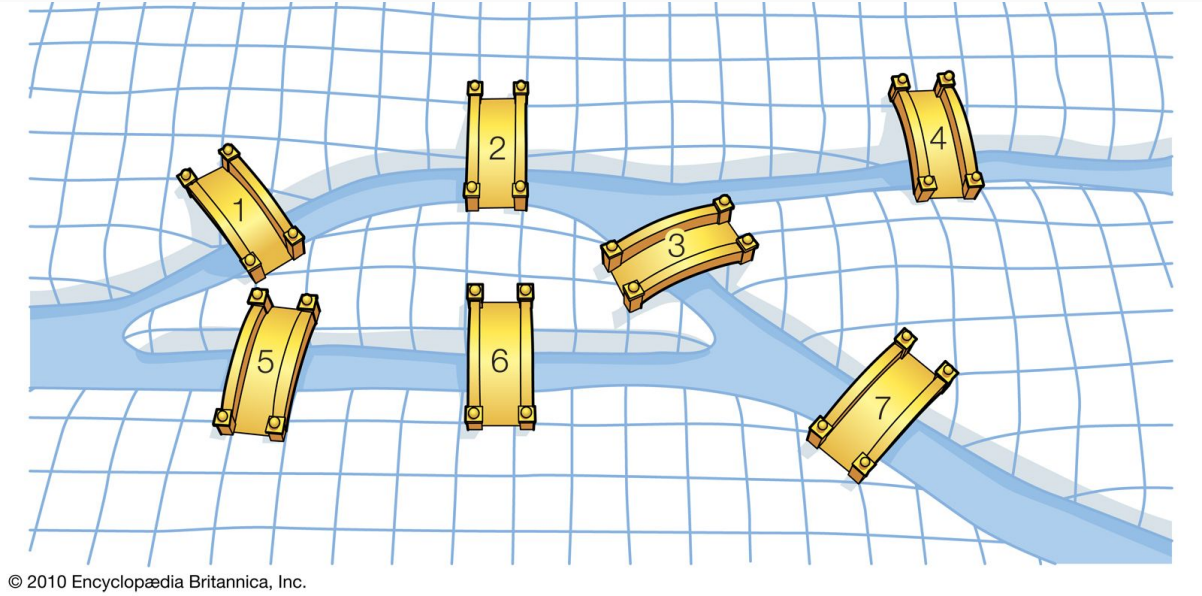
Neural Networks, an important part of modern AI, uses many nodes and edges



We will talk more in detail about neural networks in the future.....

# Famous graph problems

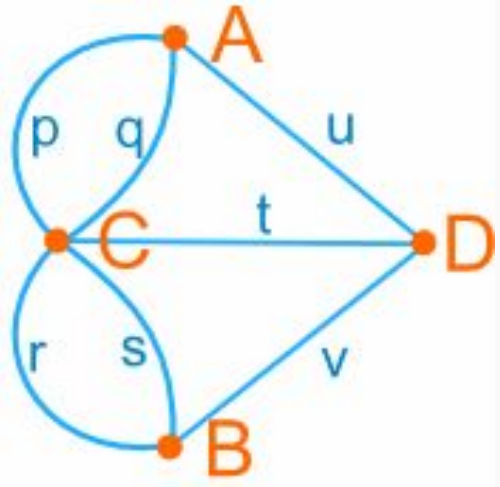
Can we go through all the bridges exactly once and end up where we started?



# Famous graph problems

For every vertex, we need to account for the bridge we use to enter and leave

This means we need only vertices with even edge count



# Famous graph problems

**Four color theorem**: To color any map region, we only need 4 different colors.

# Graphs

What things could you model with graphs?

# Quiz: http://bit.ly/FCA_Quiz_AI

# Project - Shortest route between cities

# Dijkstra's algorithm

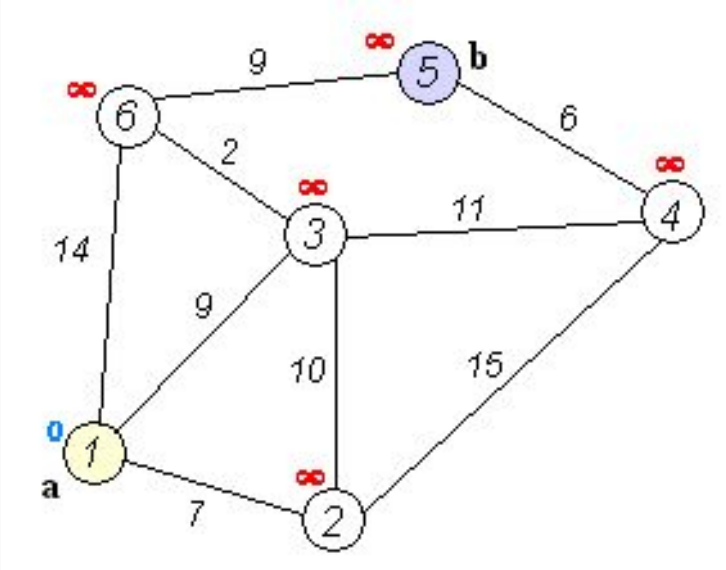
How to get from Taipei to Tainan?

This way?

This way is faster!

How can we know?

# Dijkstra's algorithm



# Dijkstra's algorithm

Dijkstra's algorithm is another way to find the shortest path between two nodes:

1. For each node, calculate the distance of getting to each of its neighbors.
1. If going through this node takes less, update our distance list.

# What information do we need?

**`Queue:`** Keep the list of nodes in our graph which we still need to update.

**`Self.distance:`** The shortest distance required to get to each city.

**`Self.parent:`** Which city do we pass through to get here?

# Get the starter file

http://bit.ly/FCA_AI2_Starter

# Let’s handle the user input

# Get the cities and weights

```
89          graph = Graph()
90          while True:
91              src = input('\nSource city: ').lower()
92              if src == 'continue':
93                  break
94              dst = input('Destination city: ').lower()
95              if dst == 'continue':
96                  break
97
98              try:
99                  weight = int(input('Weight: '))
100             except:
101                 print('Remember that the weight should be an integer! Start over')
102                 continue
103
```

# Let's add the edges with weights

```
98          try:
99              weight = int(input('Weight: '))
100         except:
101             print('Remember that the weight should be an integer! Start over')
102             continue
103
104         # TODO: add the edge between the two cities in our graph
105         graph.add_edge(src, dst, weight)
106
```

# Now let’s code the algorithm!

# Store necessary data

**Queue** holds the nodes we are exploring

**Distance** holds the shortest distance from the source to every node

**Parent** stores the parent of every node, we use it to print the path

```
26      def shortest_path(self, src, dst):
27
28          # TODO: Create vertex list
29          queue = []
30          # TODO: Make a dictionary containing the shortest distance to each city
31          self.distance = {}
32          # TODO: Store the parents of each node in a dictionary
33          self.parent = {}
```

# Do some initial setup

At the beginning, our nodes are at an “infinite” distance away.

The nodes also have no parents yet

```
35          for vertex in self.graph.keys():
36              # TODO: set all the distances to infinity
37              self.distance[vertex] = float('inf')
38              # TODO: set all the parents to None
39              self.parent[vertex] = None
40              # Add all the nodes to our initial list
41              queue.append(vertex)
```

# Set source node distance

The source node will be a distance 0 from itself

```
43          # TODO: set the distance to the initial node to 0
44          self.distance[src] = 0
```

# Find alternate distance

Try and find if the current distance to the node is better

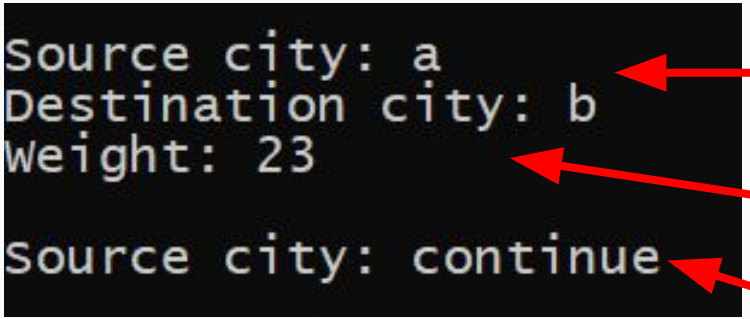
```
55                # TODO: See if we can find a path to neighbor with lower cost
56                tmp_dist = self.distance[current] + weight
57
58                # Adjust the new path if we find one
59                if tmp_dist < self.distance[neighbor]:
```

# Change the distance

If we find a shorter distance between two cities, we update our distance graph.

```
58                  # Adjust the new path if we find one
59                  if tmp_dist < self.distance[neighbor]:
60                      # TODO: Set the distance to the neighbor to a lower value if we find one
61                      self.distance[neighbor] = tmp_dist
62                      # TODO: Set the parent of the neighbor to the current node
63                      self.parent[neighbor] = current
64
```

# Go ahead and try it!




Choose the name of your cities

Distance between cities.

Type “continue” to move to next step

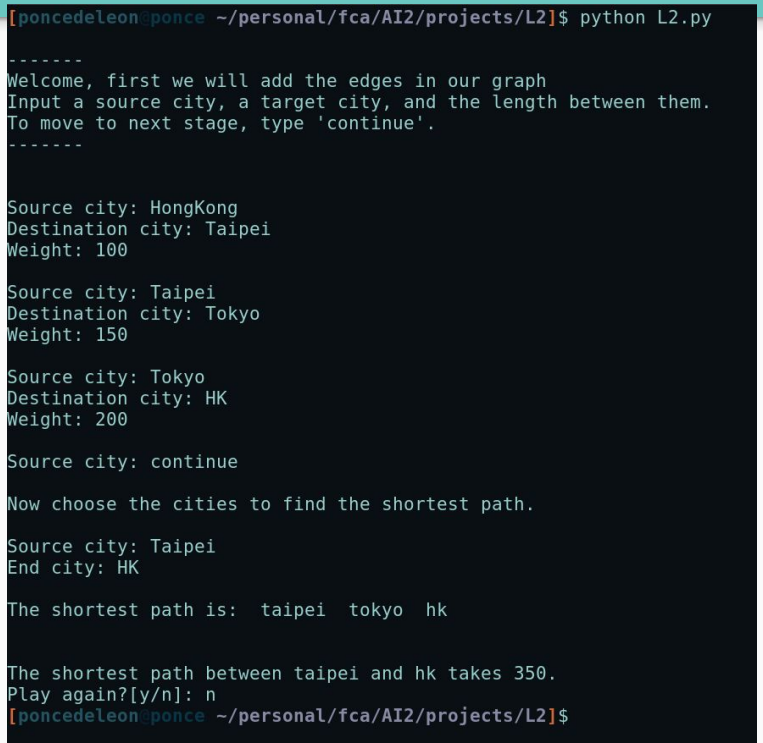
# Go ahead and try it!

Enter the two cities for which you wish to find the shortest distance.

```
Now choose the cities to find the shortest path.
Source city: a
End city: b
```

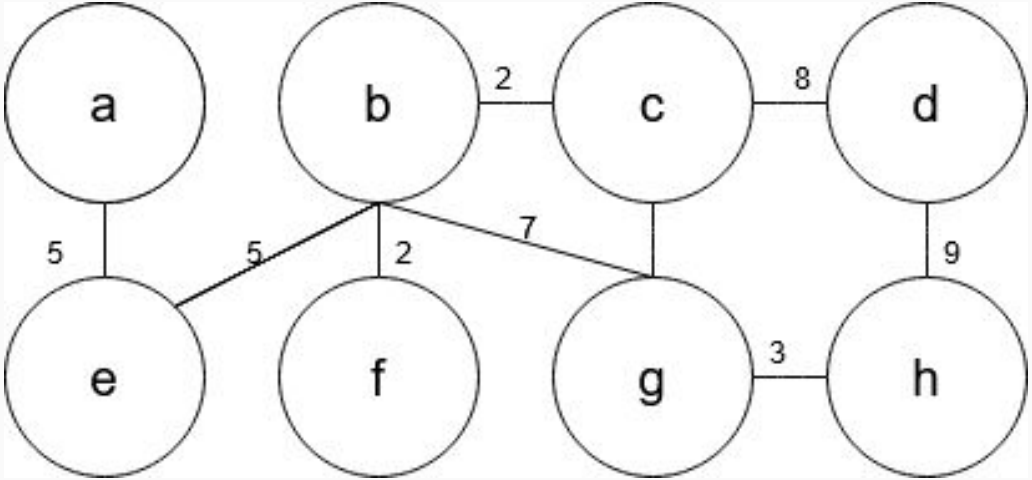
# Sample

A run of the program should resemble something like this



# Challenge

Can you make this graph? What is the shortest path from a to h?



# That’s it for today!

# Artificial Intelligence II

Lesson 2 - Graphs and Matrices

First Code Academy - Creator