CS166 Problem Set 2

https://github.com/cesar-ca/cs166-modeling-and-analysis

https://colab.research.google.com/drive/1cmP7nR74TukWbo4SsJWPioBF88b4Bn1X?usp=sharing

Problem 1: Rainfall Records

- LOs: #Modeling, #EmpiricalAnalysis
- HCs: #distributions

We explore how record rainfalls (highest ever or lowest ever) are distributed.

1. Do a bit of research online (cite your sources) and come up with a plausible continuous probability distribution for the annual rainfall (mm per year) in Berlin. Motivate your choice of distribution.

A plausible continuous probability distribution for the annual rainfall (mm per year) in Berlin is a normal distribution. On average, Berlin obtains 591 mm of rainfall per year based on data from ClimaTemps. Another source, Climate-Data, presents data (perhaps from a different year) that shows that the rainfall in Berlin is 669 mm per year.

The climate graph from Climate-Data suggests that on a month to month basis rainfall is normally distributed. Considering this phenomenon would affect yearly rainfall, then we can expect that it too would be normally distributed.

Considering the information and data found on the resources cited, a plausible continuous probability distribution for the annual rainfall (mm per year) in Berlin is a Normal distribution with a mean of 630 and a standard deviation of 26.

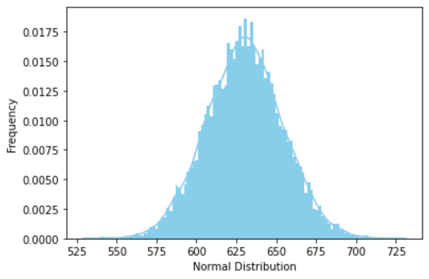
In [1]:
```python
import seaborn as sns
```

In [2]:
```python
from scipy.stats import norm

data_normal = norm.rvs(size = 10000, loc = 630, scale = 24)
```

In [3]:
```python
ax = sns.distplot(data_normal, bins = 100,
                  kde = True, color = 'skyblue',
                  hist_kws = {"linewidth":15, 'alpha':1})

ax.set(xlabel = 'Normal Distribution', ylabel = 'Frequency')
```

```
/usr/local/lib/python3.9/site-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your co
de to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

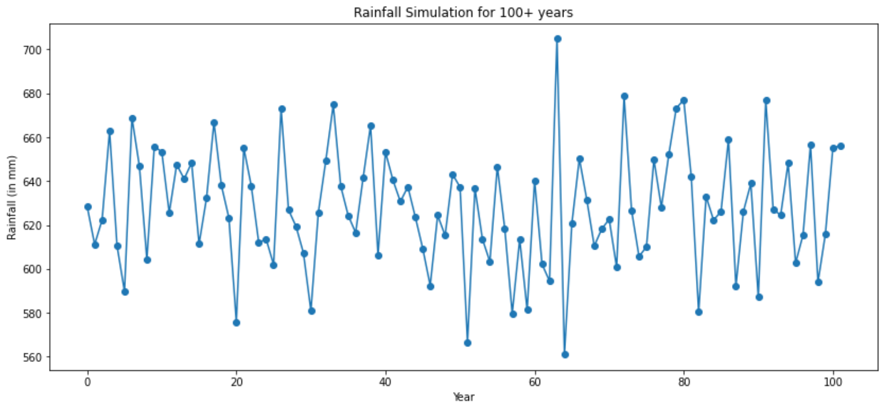
Out[3]:
```
[Text(0.5, 0, 'Normal Distribution'), Text(0, 0.5, 'Frequency')]
```



1. Next, simulate the annual rainfall in Berlin for 101 consecutive years (labeled Year 0 up to Year 100) by generating i.i.d. samples from your chosen probability distribution.

In [4]:
```python
import math as mt
import numpy as np
import matplotlib.pyplot as plt


def rainfallSimulation():
    number_list = []

    for i in range(102):
        number = round(np.random.normal(630, 26), 2)
        number_list.append((i, number))

    x = [x[0] for x in number_list]
    y = [x[1] for x in number_list]

    return (x,y)
```

In [5]:
```python
plt.rcParams["figure.figsize"] = [14.00, 6.00]
plt.plot(rainfallSimulation()[0], rainfallSimulation()[1], marker="o")
plt.title('Rainfall Simulation for 100+ years')
plt.ylabel('Rainfall (in mm)')
plt.xlabel('Year')
plt.show()
```



A rainfall value is a record high if it is greater than those in all previous years (starting with Year 0), and a record low if it is lower than those in all previous years. In the century starting from Year 1 and ending with Year 100 (including those two years), produce a histogram to show the distribution over the number of "record" years – i.e., the number of years that have **either** a record high **or** a record low.

Show empirically that the expected value of this distribution is

$$2\sum_{i=1}^{100}\frac{1}{i+1}\approx 8.395$$

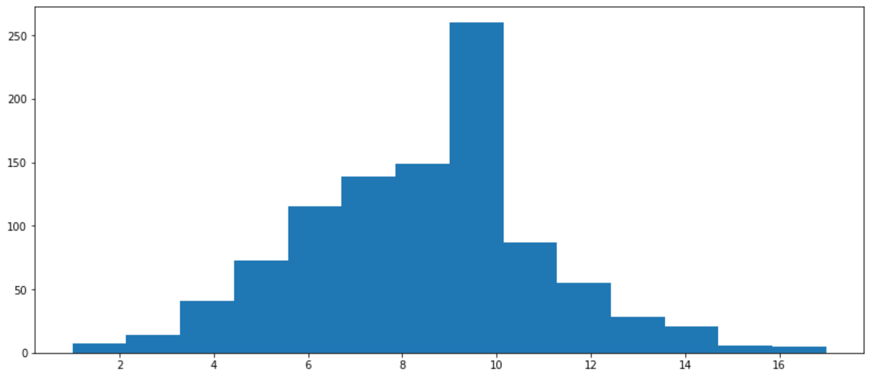
In [6]:
```python
def numberOfRecordHigh(arr):

    s = set()
    record_high = []
    number_of_high = 0

    for i in range(102):

        it = [x for x in s if x >= arr[i]]

        if len(it) == 0:
            record_high.append(-1)
            number_of_high += 1

        else:
            record_high.append(min(it))

        s.add(arr[i])

    # The variable number_of_record counts the first year, so we subtract it
    return(number_of_high - 1)
```

In [7]:
```python
def numberOfRecordLow(arr):

    S = []
    record_low = []
    number_of_low = 0

    for i in range(102):

        while (len(S) > 0 and S[-1] >= arr[i]):
            S.pop()

        if (len(S) == 0):
            record_low.append(-1)
            number_of_low += 1

        else:
            record_low.append(S[-1])

        S.append(arr[i])

    return(number_of_low - 1)
```

In [8]:
```python
high_low = []

for i in range(1000):
    high_low.append(numberOfRecordHigh(rainfallSimulation()[1]) + numberOfRecordLow(rainfallSimulation()[1]))
```

In [9]:
```python
plt.hist(high_low, bins=14)
```

Out[9]:
```
(array([  7.,  14.,  41.,  73., 115., 139., 149., 260.,  87.,  55.,  28.,
         21.,   6.,   5.]),
 array([ 1.        ,  2.14285714,  3.28571429,  4.42857143,  5.57142857,
         6.71428571,  7.85714286,  9.        , 10.14285714, 11.28571429,
        12.42857143, 13.57142857, 14.71428571, 15.85714286, 17.        ]),
 <BarContainer object of 14 artists>)
```



1. Next, use your rainfall distribution to simulate the first year after Year 0 when a record high (not low) rainfall occurs. Produce a histogram of the distribution over the number of years until the first record high after Year 0.

   Show empirically, using a line plot on top of your histogram, that this distribution has probability mass function

$$P(k)=\frac{1}{k\,(k+1)}$$

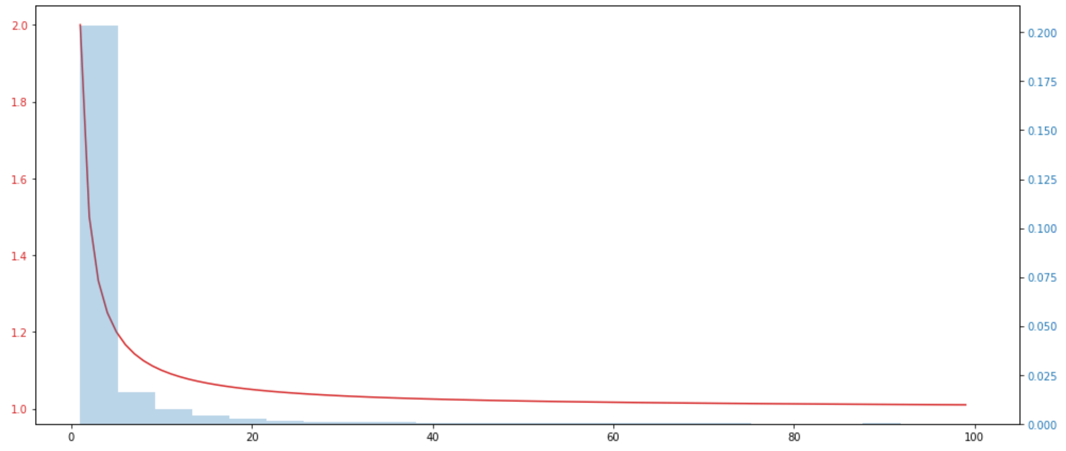
In [10]:
```python
def firstYearOfRecordHigh(arr):

    s = set()
    year_of_record_high = []
    first_year = []

    for i in range(102):
        it = [x for x in s if x >= arr[i]]

        if len(it) == 0:
            year_of_record_high.append(-1)

        else:
            year_of_record_high.append(min(it))

        s.add(arr[i])
    year_of_record_high[0] = 0

    for i in range(102):
        if year_of_record_high[i] == -1:
            first_year.append(i)

    return(first_year[0])
```

In [11]:
```
high_year = []

for i in range(10000):
    try:
        high_year.append(firstYearOfRecordHigh(rainfallSimulation()[1]))
    except IndexError:
        pass
```

In [12]:
```
def f(k):
    try:
        return 1 / k * (k + 1)
    except RuntimeWarning:
        pass

t1 = np.arange(0.0, 100.0, 1)
```

In [13]:
```
import numpy as np
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.plot(t1, f(t1), color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.hist(high_year, density = True, bins=24, alpha=0.3)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```

```
<ipython-input-12-0aec276101b2>:3: RuntimeWarning: divide by zero encountered in true_divide
  return 1 / k * (k + 1)
```



1. Explain why your results in Questions 2 and 3 do **not** depend on your choice of rainfall distribution in Question 1.

The results in Questions 2 and 3 do not depend on the choice of rainfall distribution in Question 1 because esentially we are sampling randomly from a given probability distribution and because of the central limit theorem we ought to find that the samples follow a Normal distribution. Also, given that we choose a mean for the probability distribution, a significant portion of the samples will rapidly be the year in which there is an all-time high and it becomes less likely that it will take a lot of time for a year to have an all-time high.

Problem 2: Metropolis-Hastings

- LOs: #PythonImplementation, #CodeReadability

Your task is to generate samples from a 2-dimensional probability distribution using the Metropolis-Hastings sampling algorithm.

- Implement the Metropolis-Hastings algorithm.
- Explain what your proposal distribution is and why you chose it.
- Explain the output from the algorithm, including how long it takes to reach equilibrium and how many steps you take between samples (and why).

The normalization constant of the distribution is not known. Use the Python function below to compute the unnormalized distribution.

```
def unnormalized_distribution(x, y):

    from numpy import arctan2, pi
    from scipy.stats import norm

    measurements = [2.349080, -1.276010, -2.081865]
    buoys = [[-2.5, 2], [2, -5], [-3, -4]]
    sigma = 10/180*pi
    posterior = 1

    for i in range(3):
        bearing = arctan2(buoys[i][1] - y, buoys[i][0] - x)
        difference = (bearing - measurements[i] + pi) % (2 * pi) - pi
        posterior *= norm.pdf(difference, loc=0, scale=sigma)

    return posterior
```

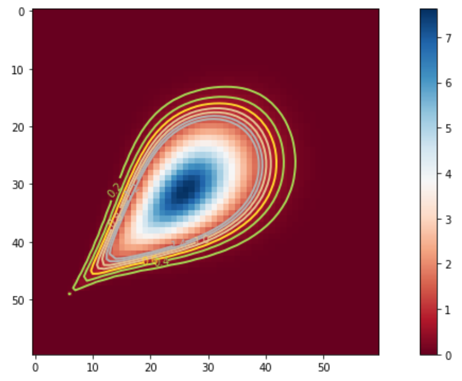
For context, this distribution arises from the following Bayesian inference problem.

You are on a boat and measure your bearings (directions) to 3 buoys. Your measurements are not very accurate and the error in the measured direction in degrees is $\text{Normal}(0, 10^2)$ – so a standard deviation of 10 degrees. Use the known locations of the buoys and your 3 noisy measurements to determine where you are on the sea.

The figure to the right shows the locations of the 3 buoys, the noisy directions measured from the location of the boat to those buoys, and the posterior distribution over the boat's location (calculated by the Python function above) from which you need to generate samples.

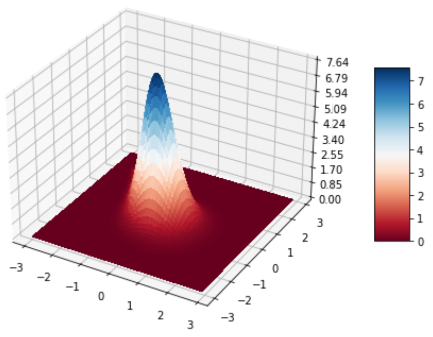
In [14]:

```python
from numpy import exp,arange
from pylab import meshgrid,cm,imshow,contour,clabel,colorbar,axis,title,show

# here target_distribution is the given unnormalized_distribution
def target_distribution(x, y):

    from numpy import arctan2, pi
    from scipy.stats import norm

    measurements = [2.349080, -1.276010, -2.081865]
    buoys = [[-2.5, 2], [2, -5], [-3, -4]]
    sigma = 10/180*pi
    posterior = 1

    for i in range(3):
        bearing = arctan2(buoys[i][1] - y, buoys[i][0] - x)
        difference = (bearing - measurements[i] + pi) % (2 * pi) - pi
        posterior *= norm.pdf(difference, loc=0, scale=sigma)

    return posterior

x = arange(-3.0,3.0,0.1)
y = arange(-3.0,3.0,0.1)

X,Y = meshgrid(x, y)
Z = target_distribution(X, Y) # evaluates the target distribution on the grid

im = imshow(Z,cmap=cm.RdBu) # drawing the function

# adds Contour lines with labels
cset = contour(Z,arange(-1,1.5,0.2),linewidths=2,cmap=cm.Set2)

clabel(cset,inline=True,fmt='%1.1f',fontsize=10)

colorbar(im) # adds colobar on the right

show()
```



In [15]:

```python
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca(projection = '3d')
surf = ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1, cmap = cm.RdBu, linewidth = 0, antialiased = False)

ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```



In [16]:

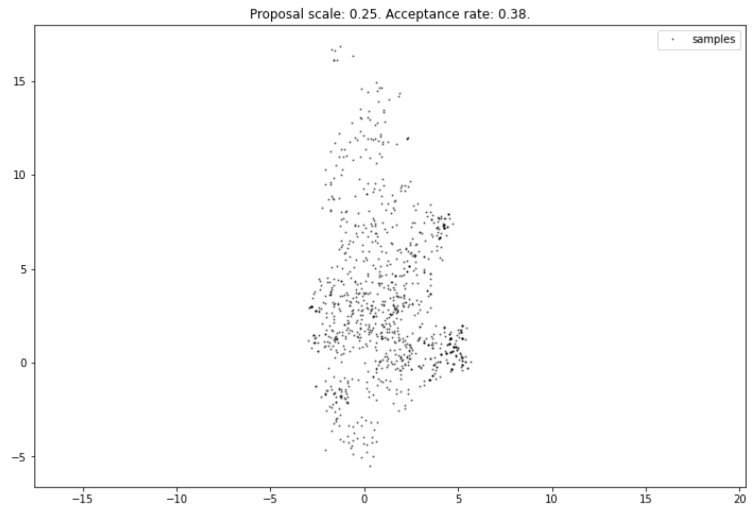
```python
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as sts

def target_density(x):

    from numpy import arctan2, pi
    from scipy.stats import norm

    y = np.random.normal(0, 2)

    measurements = [2.349080, -1.276010, -2.081865]
    buoys = [[-2.5, 2], [2, -5], [-3, -4]]
    sigma = 10/180*pi
    posterior = 1

    for i in range(3):
        bearing = arctan2(buoys[i][1] - y, buoys[i][0] - x)
        difference = (bearing - measurements[i] + pi) % (2 * pi) - pi
        posterior *= norm.pdf(difference, loc=0, scale=sigma)

    return posterior
```

In [17]:

```
def proposal_function(x, scale):
    return [np.random.normal(x[0], scale), np.random.normal(x[1], scale)]
```

In [22]:

```
proposal_scale = 0.25  # positive real number

total_samples = 1000
steps_between_samples = 10
total_steps = total_samples * steps_between_samples

state = np.empty((total_steps + 1, 2))
state[0,:] = [0,0]
accepted = 0

for step in range(total_steps):
    proposal = proposal_function(state[step], scale = proposal_scale)
    p = min(1, target_density(proposal[0]) / target_density(state[step].any()))
    if np.random.uniform(0, 1) < p:
        state[step + 1] = proposal
        accepted += 1
    else:
        state[step + 1] = state[step]

plt.figure(figsize = (12, 8))

plt.plot(state[1::steps_between_samples, 0],
         state[1::steps_between_samples, 1],
         'k.', markersize = 2, alpha = 0.5,
         label = 'samples')

plt.title(f'Proposal scale: {proposal_scale}. Acceptance rate: {accepted / total_steps:.2f}.')
plt.axis('equal')
plt.legend()
plt.show()
```



Diameter and Average Distance

- LOs: #TheoreticalAnalysis

When we think about a single aggregate measure to summarize the distance between all the nodes in a graph, there are two natural quantities that come to mind. One is the *diameter*, defined as the maximum (shortest) distance between any pair of nodes in the graph. Another is the *average distance*, which is the average (shortest) distance between all pairs of nodes in the graph.

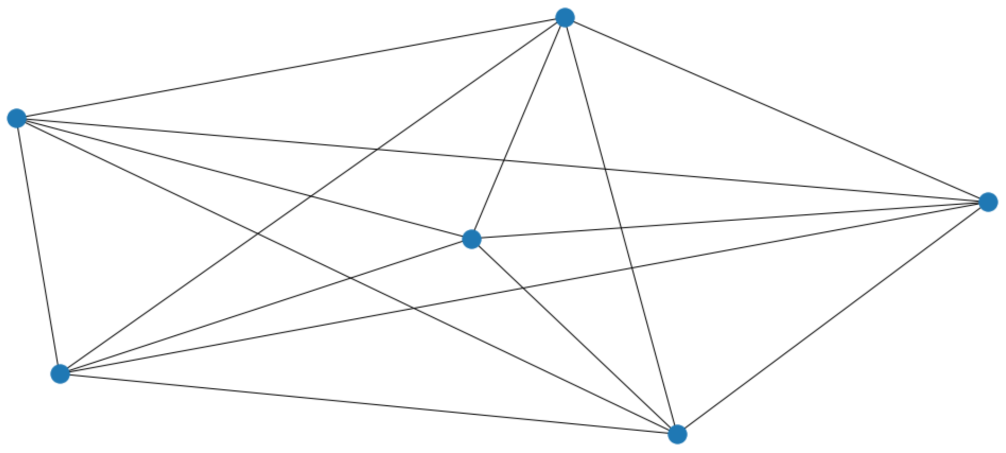
In many graphs, these two quantities are close to each other in value – but there are exceptions.

1. Provide an example of a graph with at least 6 nodes where the diameter is equal to the average distance. Explain why the metrics are equal.

The diameter of a graph describes the longest shortest path in the graph between two nodes in the graph. An example of a graph with at least 6 nodes where the diameter is equal to the average distance is a fully connected graph with 6 nodes. That is because the average distance between one node and the other is 1 and the diameter is also 1.

In [19]:

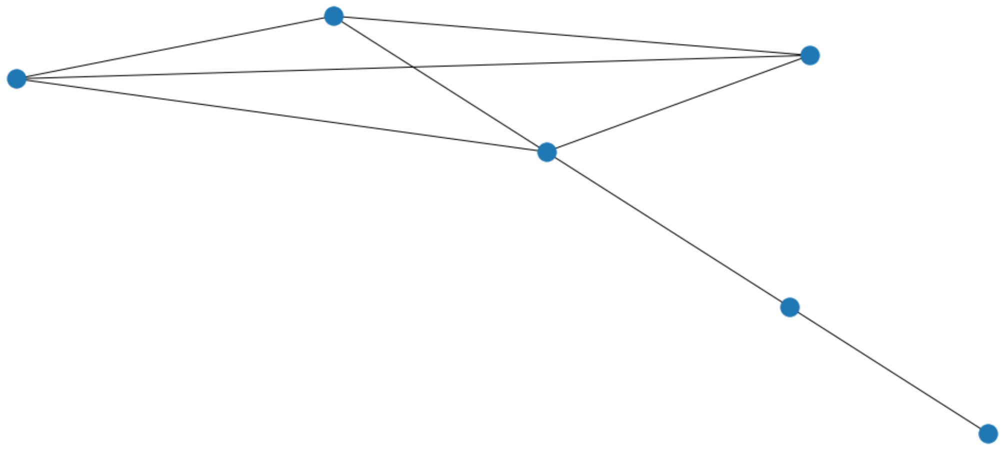
```
# importing networkx
import networkx as nx
# importing matplotlib.pyplot
import matplotlib.pyplot as plt

g = nx.Graph()

for i in range(1, 7):
    g.add_edge(i, 1)
    g.add_edge(i, 2)
    g.add_edge(i, 3)
    g.add_edge(i, 4)
    g.add_edge(i, 5)
    g.add_edge(i, 6)

nx.draw(g)
```



1. Describe an example of a graph with at least 6 nodes where the diameter is more than 3 times as large as the average distance. Explain why the diameter is more than 3 times the average distance.

An example of a graph with at least 6 nodes where the diameter is more than 3 times as large as the average distance would be one where a few (namely 3 or 4) nodes form a loop and/or a complete graph and all the other nodes are connected as if in a tail to one of the nodes in the loop. This will cause the longest shortest path to increase faster than the average distance which by virtue of the loop is closer to 1 and the nodes outside the loop help the diameter grow.

```
In [20]:
g = nx.Graph()

g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(1, 4)
g.add_edge(1, 5)
g.add_edge(5, 6)
g.add_edge(1, 3)
g.add_edge(2, 4)

nx.draw(g)
```



1. Describe how you can construct a graph in which the diameter is more than c times as large as the average distance – for any positive value of c. Briefly explain why or prove that the diameter is more than c times the average distance.

To construct a graph in which the diameter is more than c times as large as the average distance then for any positive value of c then we do something similar as before where we have a certain number of nodes which form a loop/complete graph and then additional nodes are added as if a tail connected to one of the nodes. With a sufficiently large number of nodes in the complete graph and a long tail of nodes (namely c nodes) this satisfies the requirement.

Hopfield Networks

- LOs: #PythonImplementation, #CodeReadability

A *Hopfield network* is a model that can recover memorized binary state patterns from noisy initial conditions. If a pattern of nodes was memorized and you initialize the network with a similar pattern of nodes (but has some of the node values flipped from –1 to 1 or *vice versa*), the network will recover the memorized pattern.

This was one of the early advances in neural network research and the principles of Hopfield networks – described below – are still used today in various computational neural models.

Here are the typical assumptions made in the Hopfield network model:

- Nodes take on one of two values - either -1 or $+1$.
- The network is fully connected - every node is connected by an edge to every other node (except itself).
- The edges have weights and the weights are **not** updated while the simulation runs.
- Node states are updated synchronously in discrete time steps according to this rule:

$$s_i(t+1) = \text{sign}\left(\sum_{j \neq i} w_{ij}\, s_j\,(t)\right)$$

- Here, $s_i(t)$ is the state of node i at time t, $w_{ij} = w_{ji}$ is the synmmetric edge weight between nodes i and j, and $\text{sign}(x)$ is a function that gives $+1$ if $x > 0$ and -1 otherwise.

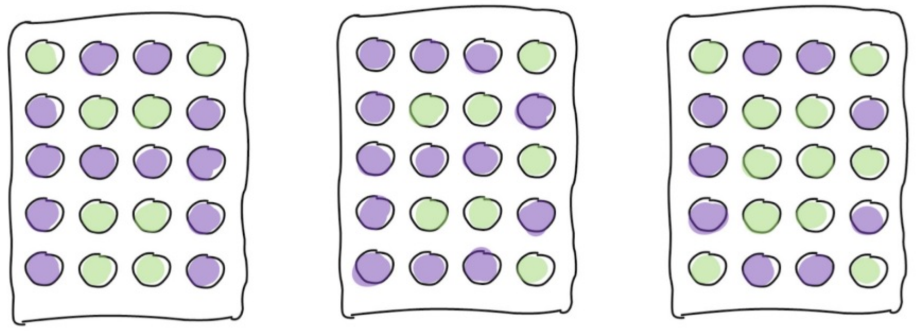
John Hopfield showed that you can store a finite number of "memories" (or patterns) in this network by carefully setting the edge weights using the following formula:

$$w_{ij} = \frac{1}{K}\sum_{k=1}^{K} S_{ik}\, S_{jk}$$

Here, s_{ik} is the state of node i in the k th pattern (of K total patterns) to be stored in the network. So if two nodes in a pattern are equal (both $+1$ or both -1), their edge weight is increased but if they are unequal, their edge weight is decreased.

Based on Hopfield's work, the capacity of a network, defined as the number of distinct memories that the network can store without getting errors, is approximately $0.138N$ where N is the number of nodes in the network. Your task is to test this with a small network and a small number of patterns.

For a network with 20 nodes, $0.138N = 2.76$ which means we should be able to store 2 patterns but not 3. Test this hypothesis using the patterns below (the first 3 letters of the Latin alphabet) and show that you can store any 2 of these patterns but not all 3.



To run your tests, start the simulation from a random initial state and show that the network ends up in one of the three patterns above after a few steps – but only if you store any 2 of these 3 patterns and not all 3.

If you run into any problems or interesting behavior in this simulation, be sure to describe your observations and explain why you think they occur.

I was unable to run the required packages for Hopfield network python implementation in Jupyter Notebook, so the answer for this is here https://colab.research.google.com/drive/1cmP7nR74TukWbo4SsJWPioBF88b4Bn1X?usp=sharing

In [21]:
```
from IPython.core.display import HTML
HTML("""
<style>

div.cell { /* Tunes the space between cells */
margin-top:1em;
margin-bottom:1em;
}

div.text_cell_render h1 { /* Main titles bigger, centered */
font-size: 2.2em;
line-height:1.4em;
text-align:center;
}

div.text_cell_render h2 { /*  Parts names nearer from text */
margin-bottom: -0.4em;
text-align:center;
}


div.text_cell_render { /* Customize text cells */
font-family: 'Times New Roman';
font-size:1.35em;
line-height:1.4em;
padding-left:3em;
padding-right:3em;
}
</style>
""")
```

Out[21]: