CSE 4632:DSP LAB 2

ID: 190041220

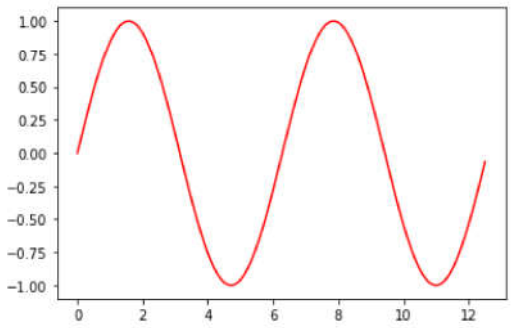
Name : Tasfia Tasneem Annesha

1.Write a code that will take a sinusoid and quantize it using **b** bits.

Use the sinusoid **x(t) = sin(t)** in the interval t=0:0.1:4*pi. Quantize x(t) using b = 4,8,12 and 16 bits. For each case of b, plot the original signal and the quantized signal.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
t = np.arange(0, 4*np.pi, 0.1)
x = np.sin(t)
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
plt.plot(t, x, 'r')
plt.show()
```



The **quantize()** function takes the t values, the signal x, and the number of bits b as input in order to quantize the given signal. The levels are created using the np.linspace function. The number of levels created is 2^b. The idea of quantizing is to iterate through all values of the signal and find the index of the minimum difference each element of the signal has with the levels. Then to move the signal to a range of 0-2^b, we shift and scale it.

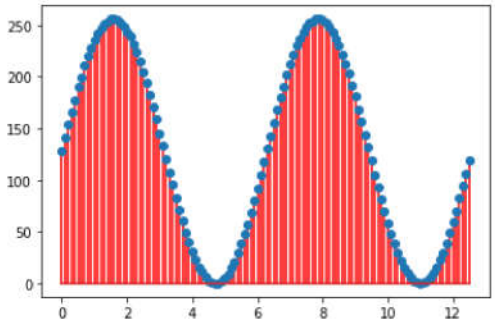
```
def quantize(t, x, b):
 levels = np.linspace(-1, 1, 2**b)
 for i in range(len(x)):
   idx = (np.abs(levels - x[i])).argmin()
   x[i] = levels[idx]
 # Shifting and scaling the signal
 x = x + 1
 x = x * ((2**b)/2)
 plt.stem(t, x, 'r')
 plt.show()
```

```
# Quantizing with b=4
x = np.sin(t)
quantize(t, x, 4)
```

```
<ipython-input-4-0826a2f90597>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inste
  plt.stem(t, x, 'r')
```

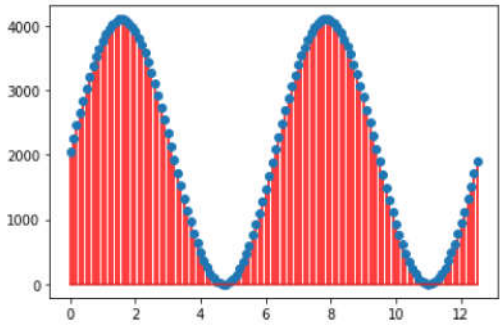

```
# Quantizing with b=8
x = np.sin(t)
quantize(t, x, 8)
```

```
<ipython-input-4-0826a2f90597>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inste
  plt.stem(t, x, 'r')
```



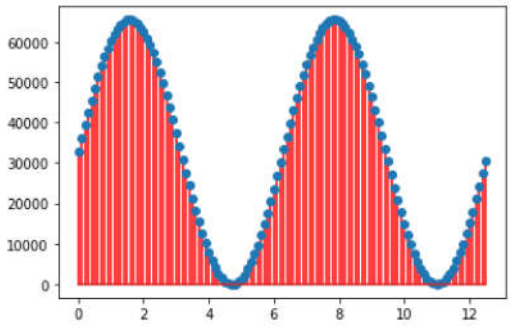

```
# Quantizing with b=12
x = np.sin(t)
quantize(t, x, 12)
```

```
<ipython-input-4-0826a2f90597>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inste
  plt.stem(t, x, 'r')
```




```
# Quantizing with b=16
x = np.sin(t)
quantize(t, x, 16)
```

```
<ipython-input-4-0826a2f90597>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inste
  plt.stem(t, x, 'r')
```



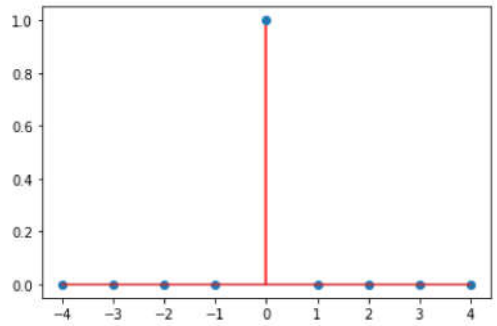
2. Write functions **delta(n)**, **unity(n)** and **unitramp(n)** which will depict the elementary signals we read about in the class. Each of this functions, for a given value of n (n>0), plots the corresponding signals in the range of −n to n.

The **delta()** function is simple. Having created an array of zeros, we just set the value of signal to 1 at index 0 (which corresponds to index n of the array x)

```
def delta(n, plot=True):
   idx = np.arange(-n, n+1, 1)
   x = np.zeros(len(idx))
   x[n] = 1
   if plot:
     plt.stem(idx, x, 'r')
     plt.show()
   return x, idx
```

```
delta(4)
```

```
<ipython-input-9-73189a437bd8>:6: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inste
  plt.stem(idx, x, 'r')
```



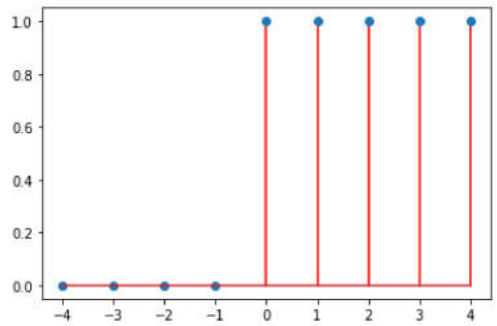

```
(array([0., 0., 0., 0., 1., 0., 0., 0., 0.]),
 array([-4, -3, -2, -1,  0,  1,  2,  3,  4]))
```

The **unity()** function uses a simple for loop to set all values to 1 starting from index 0.

```
def unity(n, plot=True):
  idx = np.arange(-n, n+1, 1)
  x = np.zeros(len(idx))
  for i in range(n, len(idx)):
    x[i] = 1
  if plot:
    plt.stem(idx, x, 'r')
    plt.show()
  return x, idx
```

```
unity(4)
```

```
<ipython-input-11-c49ebf1d3869>:7: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```




```
(array([0., 0., 0., 0., 1., 1., 1., 1., 1.]),
 array([-4, -3, -2, -1,  0,  1,  2,  3,  4]))
```

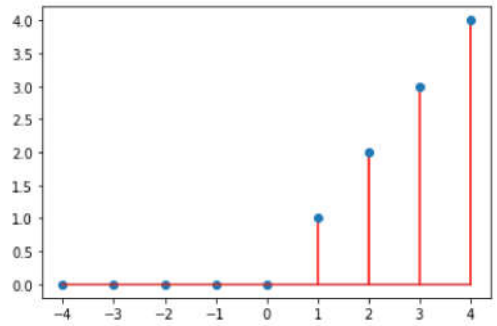
The **unitramp()** function uses the same principle as the unity() function. A for loop is used to ramp up the signal starting from 1 for all positive indices.

```
def unitramp(n, plot=True):
  idx = np.arange(-n, n+1, 1)
```

```
  x = np.zeros(len(idx))
  ramp = 0
  for i in range(n, len(idx)):
    x[i] = ramp
    ramp += 1
  if plot:
    plt.stem(idx, x, 'r')
    plt.show()
  return x, idx
```

```
unitramp(4)
```

```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```




```
(array([0., 0., 0., 0., 0., 1., 2., 3., 4.]),
 array([-4, -3, -2, -1,  0,  1,  2,  3,  4]))
```

3. Write a function that will take as input an arbitrary signal x(n) and divide it into Symmetric (**even**) and Antisymmetric (**odd**) parts and plots the three signals (original signal, even part and odd part) in the same plot.

We create the **sigfold()** function early on in order to create the signal y[n] = x[-n] which will be required to find the even and odd signals. The working princple of the sigfold() function is explained in task 5.

```
# This function is also needed in task 5
def sigfold(x):
  n = len(x)
  n = (n-1)/2
  idx = np.arange(-n, n+1, 1)
  new_idx = -idx[::-1]
  y = x[::-1]
  return y, new_idx
```

Once the folded signal of x was found, the even and odd signals were simply calculated using the formulas below:

even signal = (x[n] + x[-n]) / 2

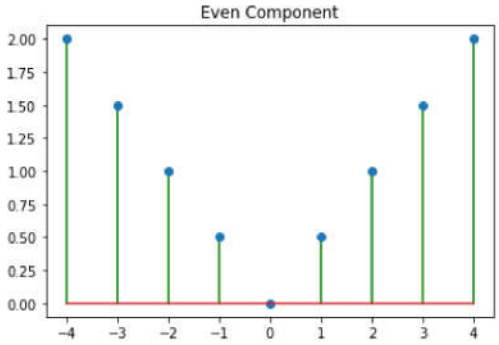
odd signal = (x[n] - x[-n]) / 2

```
x, _ = unitramp(4, plot=False)
y, idx = sigfold(x) # y = x[-n]
x_even = (x + y)/2
x_odd = (x - y)/2
plt.stem(idx, x, 'r', label='Original Signal')
plt.title("Original Signal")
plt.show()
```

```
<ipython-input-16-d004381299f0>:5: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r', label='Original Signal')
```

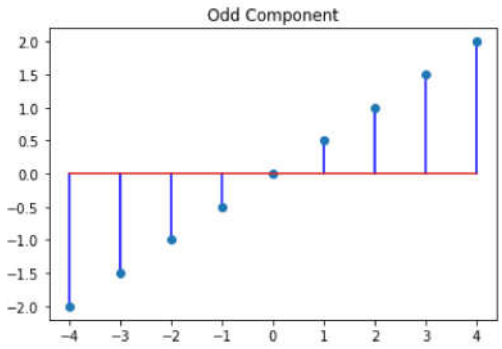

```
plt.stem(idx, x_even, 'g', label='Even Component')
plt.title("Even Component")
plt.show()
```

```
<ipython-input-17-08096d500494>:1: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x_even, 'g', label='Even Component')
```




```
plt.stem(idx, x_odd, 'b', label='Odd Component')
plt.title("Odd Component")
plt.show()
```

```
<ipython-input-18-09fdacb4d4d0>:1: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x_odd, 'b', label='Odd Component')
```

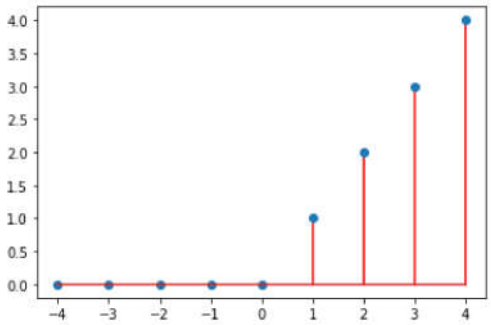


4. Write a function sigshift that takes a signal x(n) and a shift value k as inputs and returns the resulting signal y(n) = x(n-k).

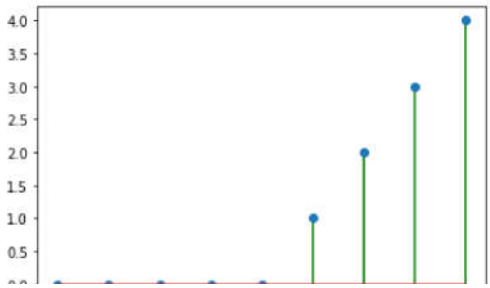
Instead of shifting the whole signal by k, it is easier and more efficient to keep the signal the same as the input but shift the x-axis values by k. This has been done in the **sigshift()**function below.

```
# The trick is to shift the labels by k to solve the problem efficiently
def sigshift(x, k):
 n = len(x)
 n = (n-1)/2
 idx = np.arange(-n, n+1, 1)
 new_idx = idx + k
 y = x
 return y, new_idx


x, _ = unitramp(4, plot=True)
y, idx = sigshift(x, 1)
plt.stem(idx, y, 'g')
plt.show()
```

```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```




```
<ipython-input-20-54e96e2a566c>:3: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, y, 'g')
```



5. Write a function sigfold that takes a signal x(n) and returns the resulting signal y(n) = x(-n). Verify the correctness of your function by taking suitable signals as input/output and plotting them.

The idea of the sigfold() function is to just reverse the signal and the corresponding indices (the sign of the indices will also change). This can be done in python using a one-liner as shown in the implementation below

```
def sigfold(x):
 n = len(x)
 n = (n-1)/2
 idx = np.arange(-n, n+1, 1)
 new_idx = -idx[::-1] # Shorthand code to reverse an array
 y = x[::-1]
 return y, new_idx


x, _ = unitramp(4, plot=True)
y, idx = sigfold(x)
plt.stem(idx, y, 'g')
plt.show()
```

```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```

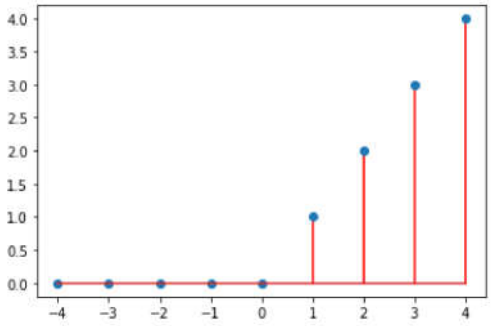
6. Write a function **downsample** that takes a signal x(n) and a value d (d is an integer, d>1) and returns the down-sampled signal y(n) = x(dn). Verify the correctness by plotting suitable input/output signals.

The basic idea of downsample() is to create the output signal y by assigning the values of x where the indeces of x is divisible by the given scalar d. We then resize the output signal to match the size of the indices

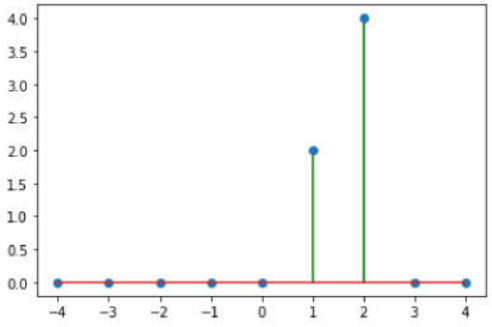
```
def downsample(x, d):
 n = len(x)
 n = (n-1)/2
 new_idx = np.arange(-n, n+1, 1)
 y = x[x%d==0]
 y.resize(new_idx.shape) # To match the indices array shape
 return y, new_idx
```

```
x, _ = unitramp(4, plot=True)
y, idx = downsample(x, 2)
plt.stem(idx, y, 'g')
plt.show()
```

```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```



```
<ipython-input-24-69526fcb15cc>:3: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, y, 'g')
```



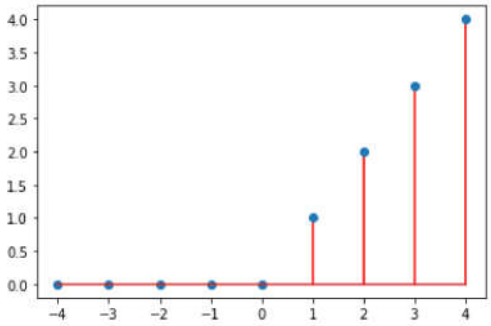
7. Write a function sigadd that takes two signals x1(n) and x2(n) as inputs and returns the resulting signal y(n) = x1(n)+x2(n).

The **sigadd()** function takes two input signals **x1 and x2**. It can handle cases where the lenght of the arrays x1 and x2 don't match. To do that we at first create a new index for our output signal ranging from the minimum value between idx1, idx2 and the maximum value between idx1, idx2. Once the output index array is created, we create two component signals y1 and y2 from the original signals x1 and x2 by assigning the non-zero values to the corresponding positions of y1 and y2 respectively. Once the component signals are created, we can get the main signal y by simply adding the components. Note: We assume that the signal does not repeat after index n.

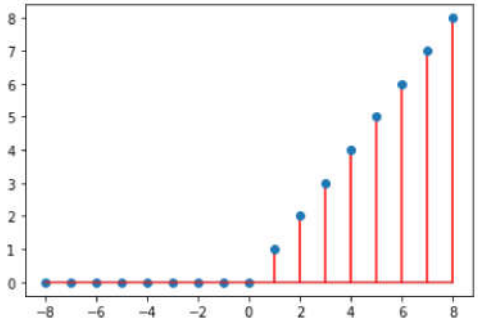
```
def sigadd(x1, x2):
 n1 = len(x1)
 n1 = (n1-1)/2
```

```
 idx1 = np.arange(-n1, n1+1, 1)
 n2 = len(x2)
 n2 = (n2-1)/2
 idx2 = np.arange(-n2, n2+1, 1)
 low = min(min(idx1), min(idx2))
 high = max(max(idx1), max(idx2)) + 1
 idx = np.arange(low, high, 1)
 y1 = np.zeros(len(idx))
 y2 = np.zeros(len(idx))
 y1[np.nonzero((idx >= min(idx1)) & (idx <= max(idx1)))] = x1
 y2[np.nonzero((idx >= min(idx2)) & (idx <= max(idx2)))] = x2
 y = y1 + y2
 return y, idx


x1, _ = unitramp(4, plot=True)
x2, _ = unitramp(8, plot=True)
y, idx = sigadd(x1, x2)
plt.stem(idx, y, 'g')
plt.title("Resultant Signal")
plt.show()
```

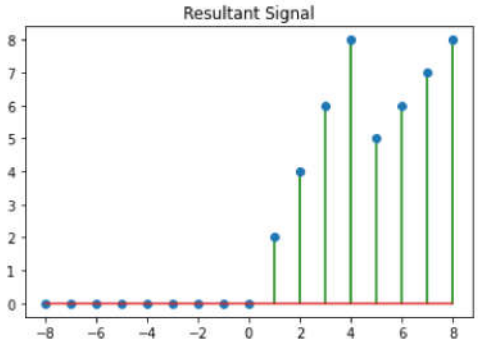
```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```



```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```



```
<ipython-input-26-509cc5417b10>:4: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, y, 'g')
```

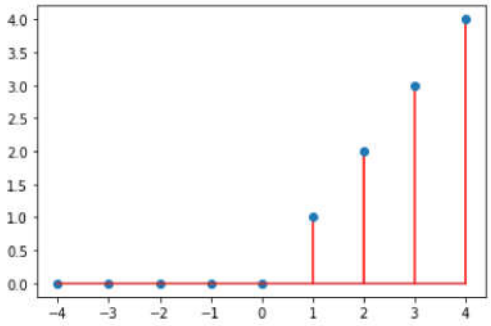


8.Write a function sigmult that takes two signals x1(n) and x2(n) as inputs and returns the resulting signal y(n) = x1(n)*x2(n).

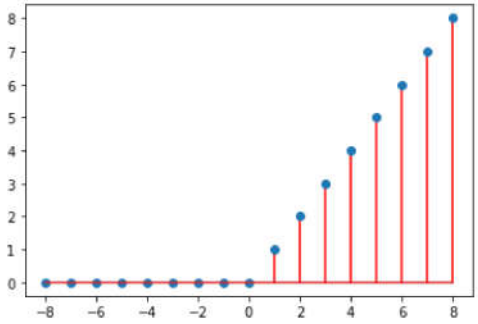
The way the sigmult() function is built is exactly the same as sigadd() in task 7. The only difference is that after getting the component singals, instead of adding them together, we multiply them to get the final output. Note: We assume that the signal does not repeat after index n.

```
def sigmult(x1, x2):
  n1 = len(x1)
  n1 = (n1-1)/2
  idx1 = np.arange(-n1, n1+1, 1)
  n2 = len(x2)
  n2 = (n2-1)/2
  idx2 = np.arange(-n2, n2+1, 1)
  low = min(min(idx1), min(idx2))
  high = max(max(idx1), max(idx2)) + 1
  idx = np.arange(low, high, 1)
  y1 = np.zeros(len(idx))
  y2 = np.zeros(len(idx))
  y1[np.nonzero((idx >= min(idx1)) & (idx <= max(idx1)))] = x1
  y2[np.nonzero((idx >= min(idx2)) & (idx <= max(idx2)))] = x2
  y = y1 * y2
  return y, idx


x1, _ = unitramp(4, plot=True)
x2, _ = unitramp(8, plot=True)
y, idx = sigmult(x1, x2)
plt.stem(idx, y, 'r')
plt.title("Resultant Signal")
plt.show()
```

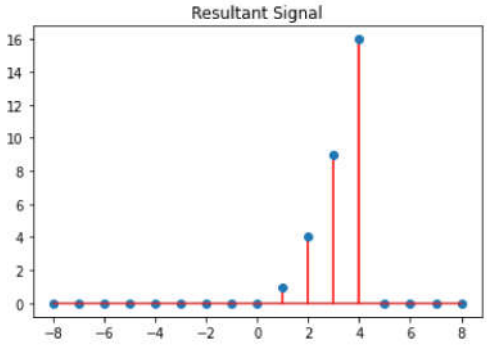
```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```



```
<ipython-input-13-b7de98cd463d>:9: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, x, 'r')
```



```
<ipython-input-28-23f932732496>:4: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection inst
  plt.stem(idx, y, 'r')
```



Write a function accumulate that takes in an input x(n) and generates the corresponding output y(n) of an accumulator as given in the equation

$$y(n) = \sum_{k=-\infty}^{n} x(k) = x(n) + x(n-1) + x(n-2) + \cdots \text{ (accumulator)}$$

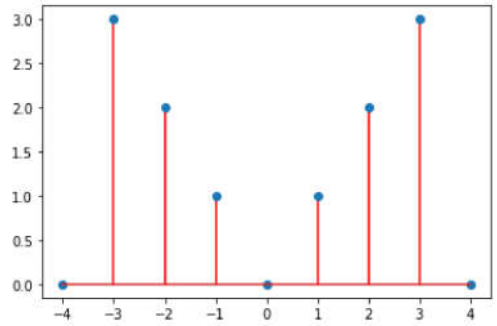
$$x(n) = \{\ldots, 0, 3, 2, 1, \underset{\uparrow}{0}, 1, 2, 3, 0, \ldots\}$$

$$y(n) = \{\ldots, 0, 3, 5, 6, \underset{\uparrow}{6}, 7, 9, 12, 0, \ldots\}$$

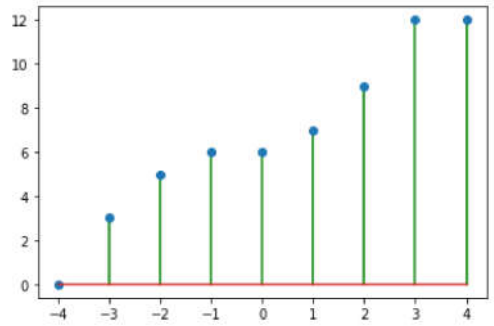
below.

```
def accumulate(x):
    n = len(x)
    y = np.zeros(n)
    boundary = int((n-1)/2)
    idx = np.arange(-boundary, boundary + 1, 1)

    y[0] = x[0]
    for i in range(1, n):
        y[i] = y[i-1] + x[i]

    return y, idx

x = [0, 3, 2, 1, 0, 1, 2, 3, 0]
boundary = int((len(x)-1)/2)
idx = np.arange(-boundary, boundary + 1, 1)

plt.stem(idx, x, linefmt = 'r')
plt.show()
y, acc_idx = accumulate(x)
plt.stem(acc_idx, y, linefmt = 'g')
plt.show()
```

```
<ipython-input-29-fce2c4d8b72c>:17: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection ins
  plt.stem(idx, x, linefmt = 'r')
```




```
<ipython-input-29-fce2c4d8b72c>:20: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection ins
  plt.stem(acc_idx, y, linefmt = 'g')
```



✓ 0s completed at 4:17 PM