18 Nov 2017, 21 Apr 2018, 27 Apr 2018, 24 Nov 2017

If a train is travelling at a certain speed and requires to stop at a particular block, it needs to start reducing its speed early in order to come to a complete stop at the block. The function brake_at(dest, speed) takes as input the block number of a destination station and the speed of the train, and returns the closest block number to the destination at which the train has to start reducing its speed by half in order to come to a complete stop at the station. Provide an implementation for the function brake_at . Assume that the train is approaching from the direction of a smaller block number.

In [1]:
```
def brake_at(dest, speed):

    counter = 0

    while speed > 0:

        if speed%2 == 0:
            speed = int(speed/2)
        else:
            speed = int((speed-1)/2)

        counter = counter + speed

    return dest - counter
```

In [2]:
```
brake_at(89, 4)
```

Out[2]:
```
86
```

In [3]:
```
brake_at(89, 10)
```

Out[3]:
```
81
```

In [4]:
```
brake_at(89, 20)
```

Out[4]:
```
71
```

The function braking_points(curr, dest, speed) takes as inputs the current and des�tination block of the train together with its current speed. It returns a sequence (either list or tuple) of block numbers where the train has to reduce its speed in order to come to a stop at the destination in the least time, i.e., the train will reduce speed at late as possible. If it is not possible to come to a stop at the destination, an empty sequence is returned.

In [5]:
```
def braking_points(curr, dest, speed):

    points = []

    if brake_at(dest, speed) < curr :

        return points


    while dest > curr :

        if brake_at(dest, speed) < curr + speed:
                if speed%2 == 0:

                    speed = int(speed/2)
                    points.append(curr)
                    curr = curr + speed

                else:

                    speed = int((speed-1)/2)
                    points.append(curr)
                    curr = curr + speed


        else:

            curr = curr + speed


    if len(points) > 0:

        points.append(dest)
        return points
    else:

        return points
```

In [6]:
```
braking_points(71, 89, 20)
```

Out[6]:
```
[71, 81, 86, 88, 89]
```

In [7]:
```
braking_points(44, 89, 10)
```

Out[7]:
```
[74, 84, 88, 89]
```

In [8]:
```
braking_points(71, 89, 22)
```

Out[8]:
```
[]
```

Implement the function num_cards which takes a non-negative integer h, and returns the number of cards needed to build a tower of height h. Your function should solve the problem computationally, i.e. using recursion or iteration and not simply use a formula.

In [9]:
```
def num_cards(height):

    if height == 1:

        return 2

    else:

        return num_cards(height-1) + 3*(height-1) + 2
```

In [10]:
```
num_cards(1)
```

Out[10]:
```
2
```

In [11]:
```
num_cards(2)
```

```
Out[11]: 7
```

```
In [12]: num_cards(3)
```

```
Out[12]: 15
```

```
In [13]: num_cards(4)
```

```
Out[13]: 26
```

Implement a function num_triangles , which takes a non-negative integer h, and returns the number of triangles that can be formed from a tower of height h. Your function should solve the problem computationally, i.e. using recursion or iteration and not simply use a formula.

```
In [14]: def summedHeight(height):

             counter = 0

             for i in range(height):

                 counter = counter + i

             return counter
```

```
In [15]: def num_triangles_upright(height):

             if height < 2 :

                 return 0

             if height == 2:

                 return 1

             else:

                 return num_triangles_upright(height-1) + summedHeight(height)
```

```
In [16]: def num_triangles_invert(height):

             if height < 2:

                 return 0

             if height == 2:

                 return 1

             else:

                 return num_triangles_invert(height-2) + summedHeight(height)
```

```
In [17]: def num_triangles(height):

             print(num_triangles_invert(height) + num_triangles_upright(height))
```

```
In [18]: for i in range(1, 10):
             num_triangles(i)
```

```
0
2
7
17
33
57
90
134
190
```

Implement the function num_triangles that takes in a non-negative integer n, and returns the number of triangles found in a Sierpinski triangle of level n. Note we count both black and white triangles as well as any larger triangles made up of smaller ones.

In [19]:
```
def num_triangles(level):

    if level == 0:

        return 1

    else:

        return 3*num_triangles(level-1) + 2
```

In [20]:
```
num_triangles(4)
```

Out[20]:
```
161
```

Implement the function area that takes in a non-negative integer n, and returns the ratio of the total area covered by the black triangles, to the largest equilateral triangle of a level n Sierpinski triangle. For example, a level 1 Sierpinski triangle, the black area makes up 0.75 of the total triangle area.

In [21]:
```
def area(level):

    if level == 0:

        return 1

    total = 0

    for i in range(1, 2**(level)+ 2**(level), 2):

        total = total + i

    return (3**level)/total
```

In [22]:
```
area(0)
```

Out[22]:
```
1
```

In [23]:
```
area(1)
```

Out[23]:
```
0.75
```

In [24]:
```
area(2)
```

Out[24]:
```
0.5625
```

```
In [25]: area(3)

Out[25]: 0.421875
```

```
In [26]: area(4)

Out[26]: 0.31640625
```

Implement the function row that takes in a non-negative integer n, and returns the nth row of the Sierpinski triangle as a list of digits 1 and 0.

```
In [27]: def nextRow(previousArray):

             newArray = []

             for i in range(len(previousArray)+1):

                 if i == 0:
                     newArray.append(1)
                 elif i == len(previousArray):
                     newArray.append(1)
                 else:
                     newArray.append(0)

             for i in range(len(previousArray)-1):

                 newArray[i+1] = (previousArray[i] + previousArray[i+1])%2

             return newArray
```

```
In [28]: def row(level):

             if level == 0:
                 return [1]
             elif level == 1:
                 return [1,1]
             else:

                 currentArray = [1,1]

                 for i in range(1, level):

                     currentArray = nextRow(currentArray)

                 return currentArray
```

```
In [29]: row(2)

Out[29]: [1, 0, 1]
```

```
In [30]: row(3)

Out[30]: [1, 1, 1, 1]
```

```
In [31]: for i in range(16):
             print(" "*(15-i), *row(i))
```

```
                1
               1 1
              1 0 1
             1 1 1 1
            1 0 0 0 1
           1 1 0 0 1 1
          1 0 1 0 1 0 1
         1 1 1 1 1 1 1 1
        1 0 0 0 0 0 0 0 1
       1 1 0 0 0 0 0 0 1 1
      1 0 1 0 0 0 0 0 1 0 1
     1 1 1 1 0 0 0 0 1 1 1 1
    1 0 0 0 1 0 0 0 1 0 0 0 1
   1 1 0 0 1 1 0 0 1 1 0 0 1 1
  1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Implement the function num_atoms(molecule) which takes in a chemical formula of a molecule, and returns the number of atoms in the molecule

In [32]:
```python
def num_atoms(molecule):

    total = 0
    i = 0

    while i < len(molecule)-1:

        nextAtom = molecule[i+1]
        currentAtom = molecule[i]

        if currentAtom.isalpha() and nextAtom.isdigit() :


            total += int(nextAtom)
            i = i+2

        else:

            total += 1
            i = i + 1

    if molecule[-1].isalpha():

        total += 1

    return total
```

In [33]:
```python
num_atoms("H2SO4")
```

Out[33]:
```
7
```

In [34]:
```python
num_atoms("CH3CH2OH")
```

Out[34]:
```
9
```

In [35]:
```python
num_atoms("NH4CO2NH2")
```

Out[35]:
```
11
```

Implement the function molar_mass(molecule, table) which takes in a molecule and a table (which is a dict) and returns the molar mass of the molecule.

In [36]:
```
table = {'H':1.008, 'B':10.81, 'C':12.011, 'N':14.007,'O':15.999, 'F':18.998, 'P':3
```

In [37]:
```
def molar_mass(molecule, table):

    total = 0
    i = 0

    while i < len(molecule)-1:

        nextAtom = molecule[i+1]
        currentAtom = molecule[i]

        if currentAtom.isalpha() and nextAtom.isdigit() :


            total += int(nextAtom)*table.get(currentAtom)
            i = i+2

        else:

            total += 1*table.get(currentAtom)
            i = i + 1

    if molecule[-1].isalpha():

        total += 1*table.get(molecule[-1])

    return total
```

In [38]:
```
molar_mass("H2SO4", table)
```

Out[38]:
```
98.072
```

In [39]:
```
molar_mass("CH3CH2OH" ,table)
```

Out[39]:
```
46.069
```

In [40]:
```
molar_mass("NH4CO2NH2", table)
```

Out[40]:
```
78.07100000000001
```