Collatz_group

December 13, 2018

1 Report 1: The Collatz Conjecture

1.1 By Joseph Ismailyan, Alex Busalacchi, Vernon Wetzell, Ryan Springer-Carter, Sam Borghese.

```
In [151]: def Collatz(n, num=3, maxSteps=1000):
              '''
              Prints the Collatz sequence beginning with the input parameter n.
              Terminates when ...
              '''
              steps = 0
              t=0
              h=0
              inc=1
              L=[]
              cycle=[]
              while(steps <= maxSteps):
                  if(n%2 == 0):
                      # then n is even, perform even integer stuff
                      n = n/2
                      #print("{0:.0f}".format(n))
                  elif(n == 1):
                      # break when n=1
                      #print("\t{0:.0f} steps".format(steps))
                      break
                  elif(steps >= maxSteps):
                      #print("max steps reached")
                      break
                  else:
                      # n is odd, perform odd integer stuff
                      n = num*n+1
                      #print("{0:.0f}".format(n))
                  steps+=1
                  L=L+[n]
              if L[0]<0:
                  tort=L[0] #tortoise starts in the beginning , hare starts
                  hare=L[2]
```

```
            while tort!=hare:
                t=t+1
                h=h+2
                tort=L[t]
                hare=L[h]
            if tort==hare:
                print("cycle is reached")
            for x in range(h):
                cycle=cycle+[L[t+x]]
            while cycle[0]!=cycle[inc]:
                inc+=1
            print(cycle[0:inc])
        return(L)
        #print(len(L))
```

In [105]:
```
Collatz(-8884**71,3,5000)
```

```
cycle is reached
[-5.0, -14.0, -7.0, -20.0, -10.0]
5001
```

In [106]:
```
###### print out how many steps are required to reach 1 from n
def halting_steps(n):
    steps = 0
    maxSteps = 1000
    while(steps <= 1000):
        if(n%2 == 0):
            # then n is even, perform even integer stuff
            n = n/2
            steps+=1
        elif(n == 1):
            # break when n=1
            print("\t{0:.0f} steps".format(steps))
            break
        elif(steps >= maxSteps):
            # break if maxSteps reached
            print("max steps reached")
            break
        else:
            # n is odd, perform odd integer stuff
            n = 3*n+1
            steps+=1
```

In [107]:
```
halting_steps(1001)
```

```
	142 steps
```

In [108]: halting_steps(1000)

```
111 steps
```

In [109]: halting_steps(2000)

```
112 steps
```

In [110]: halting_steps(16001)

```
146 steps
```

In [111]: halting_steps(16000)

```
115 steps
```

In [112]: halting_steps(2**12)

```
12 steps
```

When using 2ˆn power as a starting number, we instantly know how many steps are required because we'll reach 1 by halving 2ˆn, n times.

In [113]: halting_steps((2+1)**12)

```
133 steps
```

Idea: Make a graph with the x axis being the number n and the y axis being the number of steps to reach 1.

In [114]:

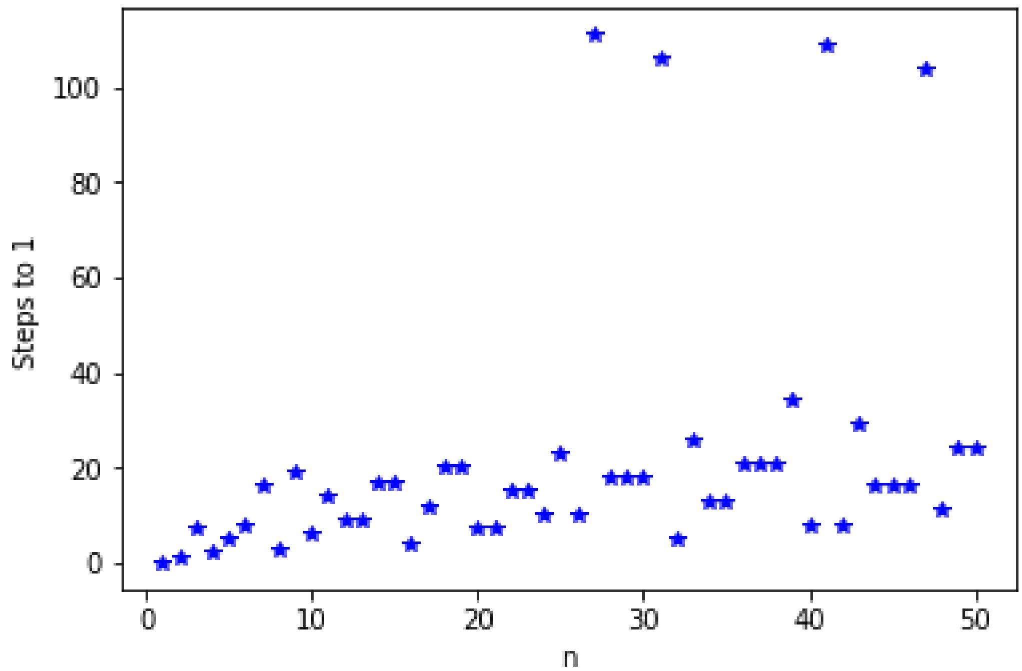
```
# this function will return the number of steps required to reach 1 instead
# of printing it. This way I can automatically test and keep track of
# the steps required for a large set of numbers.
def halting_steps_w_return(n):
    steps = 0
    maxSteps = 1000
    while(steps <= 1000):
        if(n%2 == 0):
            # then n is even, perform even integer stuff
            n = n/2
            steps+=1
        elif(n == 1):
            # break when n=1
            return steps
```

```
                break
            elif(steps >= maxSteps):
                # break if maxSteps reached
                #print("max steps reached")
                break
            else:
                # n is odd, perform odd integer stuff
                n = 3*n+1
                steps+=1
```

In [115]:
```
# write a function to automatically test and store numbers 1-n
# as well as their respective "step" counts
def collatz_test_to_n(n):
    x_axis = []
    y_axis = []
    for x in range(1,n+1):
        x_axis.append(x)
        y_axis.append(halting_steps_w_return(x))

        # it's x-1 because I started the lists at x=1 but python indices start at 0
        #print("x = {}, y = {}".format(x_axis[x-1],y_axis[x-1]))
    return x_axis, y_axis
```

In [146]:
```
# plot 'n' vs 'number of steps of n'
import matplotlib.pyplot as plt
def plot_n_vs_steps(n):
    xy_axis = collatz_test_to_n(n)
    plt.plot(xy_axis[0], xy_axis[1],'b*')
    plt.xlabel('n')
    plt.ylabel('Steps to 1')
```

In [147]:
```
# plots 'n' vs 'steps of n to reach 1' from 1 to n
plot_n_vs_steps(50)
```



I think the outliers in this data set may be prime numbers.

```
In [118]: halting_steps(27)
```

```
111 steps
```

```
In [119]: halting_steps(31)
```

```
106 steps
```

```
In [120]: halting_steps(41)
```

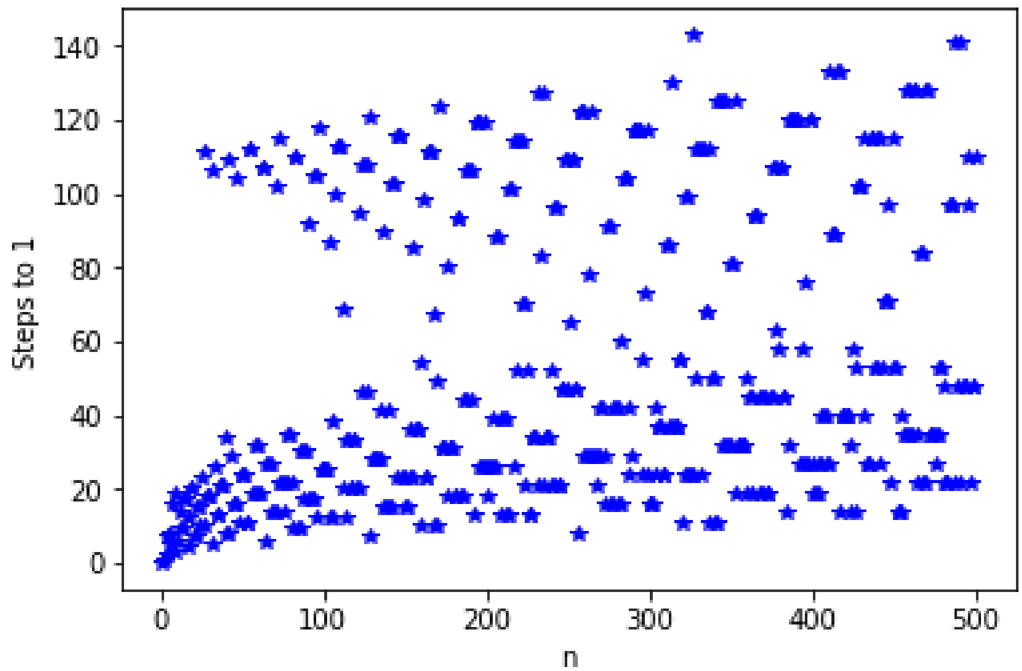
```
109 steps
```

```
In [121]: halting_steps(47)
```

```
104 steps
```

These numbers have some kind of relationship that I can't see.

```
In [148]: plot_n_vs_steps(500)
```



Idea: Plot number of steps vs prime numbers

In [123]:
```
# first I need to create a primality testing function that returns True if n is prim
from math import sqrt
def is_prime(x):
    for j in range(2,int(sqrt(x))+1):  # the list of numbers 2,3,...,n-1.
        if(x%j == 0):  # is n divisible by j?
            return False
    return True
```

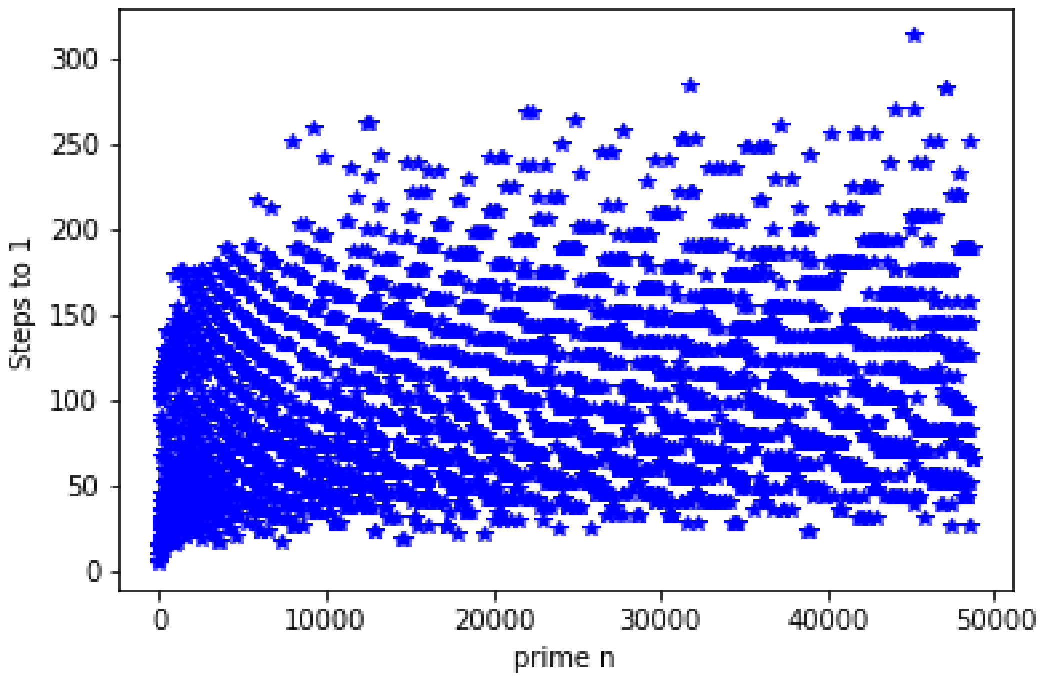
In [124]:
```
# find the first n prime numbers
def find_n_primes(n):
    primes = []
    next_num = 1

    # while I don't have enough primes, keep searching
    # I don't have error checking, be careful when using a large 'n'
    while(len(primes) < n):
        next_num = next_num + 2
        if(is_prime(next_num)):
            primes.append(next_num)
    #print(primes)
    return primes
```

In [125]:
```
# use halting_steps_w_return to return number of steps with 'primes' as input
# return array of x-values and y-values to be plotted
```

```
def collatz_test_primes(n):
    x_axis = []
    y_axis = []
    primes_list = find_n_primes(n)
    for j in range(1,n+1):
        x_axis.append(primes_list[j-1])
        y_axis.append(halting_steps_w_return(primes_list[j-1]))
    return x_axis,y_axis
```

```
In [149]: import matplotlib.pyplot as plt
          def plot_n_primes_vs_steps(n):
              xy_axis = collatz_test_primes(n)
              plt.plot(xy_axis[0], xy_axis[1], 'b*')
              plt.xlabel('prime n')
              plt.ylabel('Steps to 1')
```

```
In [150]: plot_n_primes_vs_steps(5000)
```



The highest number of steps to reach 1 of first 500 primes is higher than the number of steps to reach 1 of the first 500 natural numbers, it also seems to have a more uniform distribution of points.

```
In [128]: # find the difference between max values of
          # first 'n' natural numbers vs first 'n' prime numbers
          # and how many steps they take to reach 1
```

```
def max_steps(n):

    # recall that collatz_test_primes returns two lists, the
    # second one contains the number of steps

    # using '_, variable_name' ignores the first return
    _, prime_list = collatz_test_primes(n)
    _, regular_list = collatz_test_to_n(n)
    print(max(prime_list))
    print('average:',sum(prime_list)/500)
    print(max(regular_list))
    print('average',sum(regular_list)/500)
    print('Difference:',max(prime_list)-max(regular_list))
```

In [129]:
```
max_steps(100)
```

```
141
average: 11.502
118
average 6.284
Difference: 23
```

In [130]:
```
# compute numbers in the form 2(n^2 + n)+1
list_of_nums = []
for x in range(2, 20):
    list_of_nums.append(2*(x**2 + x)+1)
print(list_of_nums)
```

```
[13, 25, 41, 61, 85, 113, 145, 181, 221, 265, 313, 365, 421, 481, 545, 613, 685, 761]
```

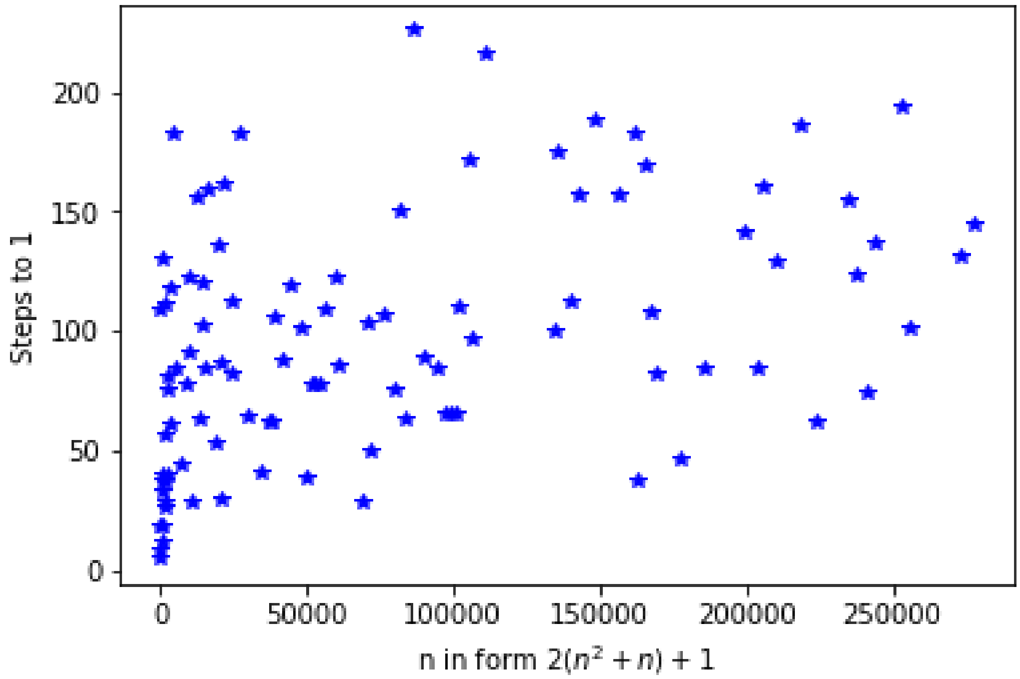
In [131]:
```
# compute prime numbers in the form 2(n^2 + n)+1
list_of_nums = []
primes_found = 0
NUM = 100
x = 1
while(primes_found < NUM):
    new_x = 2*(x**2 + x)+1
    if(is_prime(new_x)):
        list_of_nums.append(new_x)
        primes_found+=1
    x+=1
```

In [132]:
```
num_steps_for_number_in_form = []
for x in range(len(list_of_nums)):
    num_steps_for_number_in_form.append(halting_steps_w_return(list_of_nums[x]))
print(max(num_steps_for_number_in_form))
print('average', sum(num_steps_for_number_in_form)/NUM)
```

```
226
average 96.45
```

```
In [157]: plt.plot(list_of_nums, num_steps_for_number_in_form, 'b*')
          plt.xlabel(r'n in form $2(n^{2} + n) + 1$')
          plt.ylabel('Steps to 1')
```

```
Out[157]: Text(0,0.5,'Steps to 1')
```

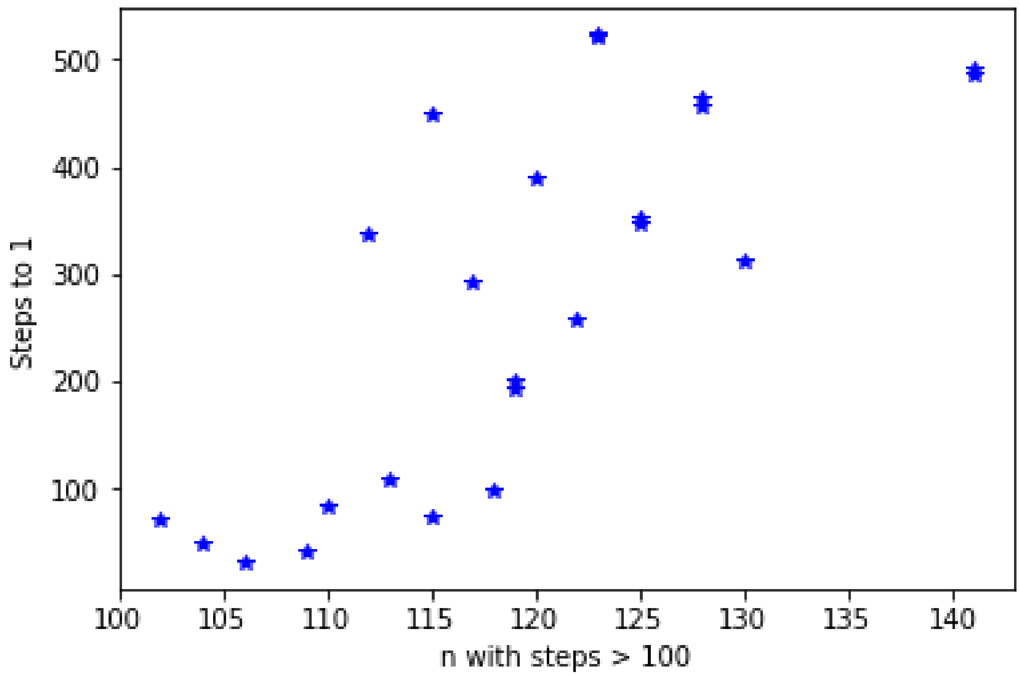



```
In [154]: # I need to find all prime numbers that require >100 steps
          # to figure out the relationship they share so
          # I might be able to predict which numbers will require more steps
```

```
In [155]: n_primes = find_n_primes(100)
          large_step_nums = []
          len_of_list = []
          for x in range(len(n_primes)):
              if(halting_steps_w_return(n_primes[x]) > 100):
                  large_step_nums.append(n_primes[x])
                  len_of_list.append(halting_steps_w_return(n_primes[x]))
          print(large_step_nums)
```

```
[31, 41, 47, 71, 73, 83, 97, 109, 193, 199, 257, 293, 313, 337, 347, 353, 389, 449, 457, 463,
```

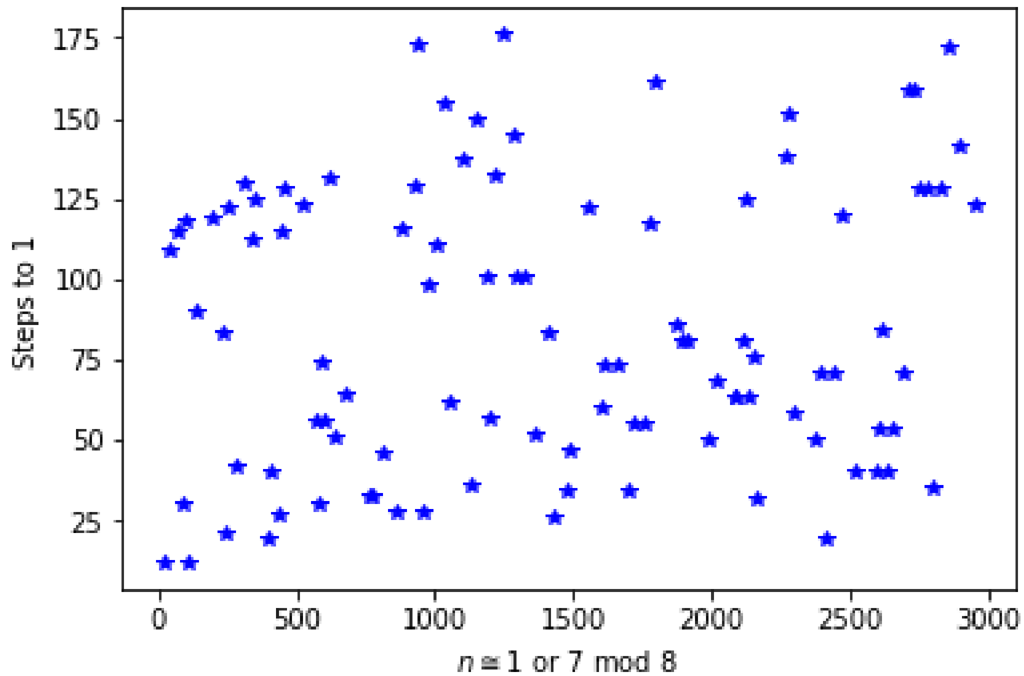
```
In [158]: plt.plot(len_of_list, large_step_nums, 'b*')
          plt.xlabel('n with steps > 100')
          plt.ylabel('Steps to 1')

Out[158]: Text(0,0.5,'Steps to 1')
```




```
In [163]: # testing this theory out
          congruent_1_7_mod_8 = []
          def find_n_primes_congruent_mod_8(n):
              primes = []
              next_num = 1

              # while I don't have enough primes, keep searching
              # I don't have error checking, be careful when using a large 'n'
              while(len(primes) < n):
                  next_num = next_num + 2
                  if(is_prime(next_num) and ((next_num%8) == (1 or 7))):
                      primes.append(next_num)
              #print(primes)
              return primes

          congruent_1_7_mod_8 = find_n_primes_congruent_mod_8(100)
          # another_damn_list will hold the # of steps for respectives primes congruent to 1,7
          another_damn_list = []
          for x in range(len(congruent_1_7_mod_8)):
```

```
            another_damn_list.append(halting_steps_w_return(congruent_1_7_mod_8[x]))
    #print(another_damn_list)
    plt.plot(congruent_1_7_mod_8, another_damn_list, 'b*')
    plt.xlabel(r'$n \cong 1$ or 7 mod 8 ')
    plt.ylabel('Steps to 1')
```

Out[163]: Text(0,0.5,'Steps to 1')



In [138]:
```
def dropping_steps(n):
    m=n
    steps = 0
    maxSteps = 10000
    while(steps <= 1000):
        if(n%2 == 0):
            # then n is even, perform even integer stuff
            n = n/2
            steps+=1
            if(m>n):
                return(steps)
                break
        elif(n == 1):
            # break when n=1
            return(steps)
            break
        elif(steps >= maxSteps):
```

```
                # break if maxSteps reached
                print("max steps reached")
                break
            else:
                # n is odd, perform odd integer stuff
                n = 3*n+1
                steps+=1
                if(m>n):
                    return(steps)
                    break
```

In [139]:
```
dropping_steps(13)
```

Out[139]: 3

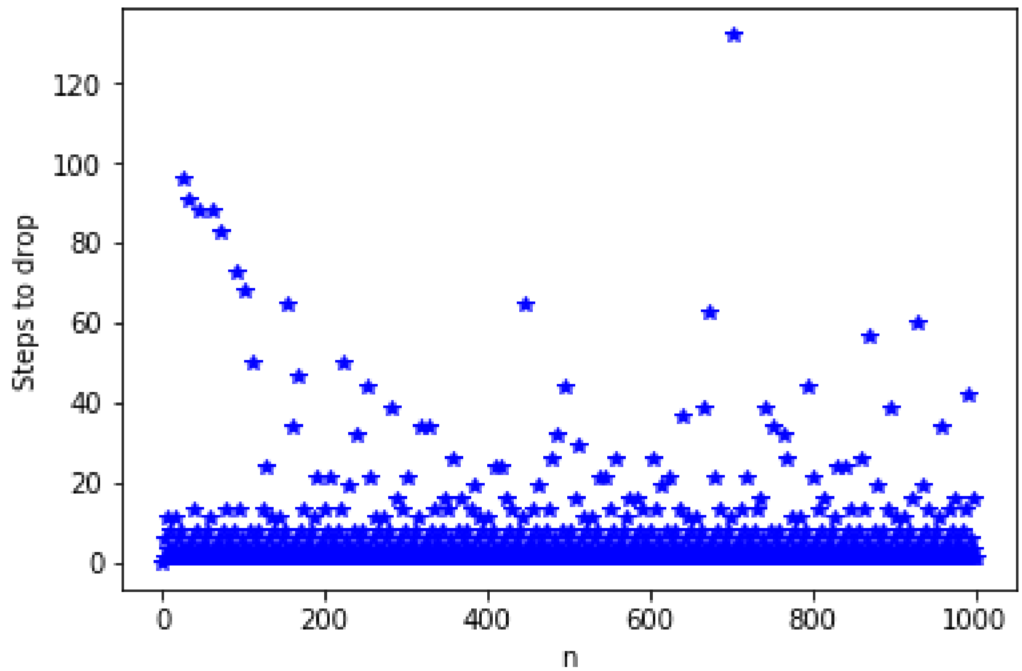
In [140]:
```
Collatz(13)
```

9

In [141]:
```
# write a function to automatically test and store numbers 1-n
# as well as their respective "step" counts
def collatz_test_to_ndrop(n, steps_to_drop):
    x_axis = []
    y_axis = []
    for x in range(1,n+1):
        x_axis.append(x)
        y_axis.append(dropping_steps(x))
        if(y_axis[x-1] > steps_to_drop):
            print(x)

        # it's x-1 because I started the lists at x=1 but python indices start at 0
        #print("x = {}, y = {}".format(x_axis[x-1],y_axis[x-1]))
    return x_axis, y_axis
```

In [164]:
```
# plot 'n' vs 'number of steps of n'
import matplotlib.pyplot as plt
def plot_n_vs_drop(n):
    xy_axis = collatz_test_to_ndrop(n, 100000)
    plt.plot(xy_axis[0], xy_axis[1], 'b*')
    plt.xlabel('n')
    plt.ylabel('Steps to drop')
```

In [165]:
```
plot_n_vs_drop(1000)
```




```
In [ ]:

In [168]: def even_odd(n):
              L = Collatz(n)
              odds = 0
              evens = 0
              for x in range(len(Collatz(n))):
                  if(L[x] % 2 == 0):
                      evens += 1
                  else:
                      odds += 1
              #print("evens: ", evens, '\n', 'odds: ', odds)
              #print(evens/odds)
              return evens/odds


In [169]: even_odd(91)

Out[169]: 1.7878787878787878

In [170]: def find_ndrop_greater_than_x(n, steps_to_drop):
              x_axis = []
              list_of_drops = []
              # first find list of drops
              for p in range(1,n):
```

```
            if(dropping_steps(p) > steps_to_drop):
                list_of_drops.append(p)
                #print(p)
        # we now have a list of drops

            # it's x-1 because I started the lists at x=1 but python indices start at 0
            # print("x = {}, y = {}".format(x_axis[x-1],y_axis[x-1]))
        return list_of_drops
```

```
In [171]: find_ndrop_greater_than_x(100, 50)
```

```
Out[171]: [27, 31, 47, 63, 71, 91]
```

```
In [175]: def graph_ratio_of_drops(n, steps_to_drop):
              x_axis = find_ndrop_greater_than_x(n, steps_to_drop)
              y_axis = []
              average = []
              total_ratio = 0
              for y in range(len(x_axis)):
                  y_axis.append(even_odd(x_axis[y]))
              plt.plot(x_axis, y_axis, 'b*')
              plt.xlabel('n with dropping steps > steps_to_drop')
              plt.ylabel('ratio of even/odd values in Collatz')

              print('max: ',max(y_axis))
              print('min: ',min(y_axis))
              #print((max(y_axis) + min(y_axis)) / 2)

              for k in range(len(x_axis)):
                  total_ratio += y_axis[k]
              print()

              avg = total_ratio/len(x_axis)
              print('average: ',avg)
              print('median: ', (max(y_axis) + min(y_axis))/2)

              for a in range(len(x_axis)):
                  average.append(avg)


              plt.plot(x_axis, average,'r')
```

```
In [185]: graph_ratio_of_drops(10000,40)
          # prints ratio of even/odd vsnumber with dropping steps greater than steps_to_drop
```

```
max:  2.210526315789474
min:  1.7073170731707317

average:  1.848991987478213
```

```
median:  1.958921694480103
```