In [1]:
```
# author: Mariya Savinov
# date: 1/31/2025
```

Installing Python

- Go to the Anaconda website https://www.anaconda.com/download/.
- Download and run the installer for your platform. You want Python 3.5 or a later version.
- See the installation instructions: https://docs.anaconda.com/anaconda/install/.
- For future reference, see the official documentation: https://docs.anaconda.com/anaconda/.
- With this, you will automatically download many packages like NumPy, Matplotlib, SciPy, pandas, TensorFlow, etc.
- Anaconda Distribution also includes applications like Jupyter Notebook, Spyder, etc. to edit, debug, and profile your programs

Running Python

- Code can be written in text editors like vim and emacs to save into a file, which you then run in a command-line window (e.g., $ python hello.py)
- Spyder is an integrated development environment (IDE) from which you can write and run code, with debugging tools
- Jupyter Notebooks are interactive documents which run in your browser, where you can combine code, text, equations, and plots in one place
- Visual Studio Code (VS Code) is another good IDE, with built-in Git support
- Both Spyder and Jupyter Notebook are installed with the Anaconda Distribution!

Example Script

At the beginning of your script, import the packages you need

In [2]:
```
import numpy as np # multidimensional arrays, mathematical functions, etc.
import matplotlib.pyplot as plt # plotting library
```

It is good practice to comment your code (using "#"), explaining what your script does and how it works.

For example, let's check if a random number is even or odd

In [3]:
```
# set x to be a random number from 0 to 100
x = np.random.randint(100)

# if-else statement to check if x is even or odd
if x % 2 == 0:
    print("x, where x={}, is even!".format(x))
```

```
else:
    print("x, where x={}, is odd!".format(x))
```

```
x, where x=86, is even!
```

Calculations

Common arithmetic **operators**:

```
+  addition
-  subtraction
*  multiplication
/  division
** power
```

Python follows the standard order of operations.

Less common arithmetic operators:

```
// integer division (divide and round DOWN)
%  modulo (remainder after division)
```

Let's make an array of three numbers and take the square root of each

In [4]:
```
y = np.array([5, 2.3, 1e2+0.345])
print(y)
y**.5
```

```
[  5.      2.3   100.345]
```

Out[4]:
```
array([ 2.23606798,  1.51657509, 10.01723515])
```

Rounding Error

1. A computer can't store an infinite number of digits.
2. Computers store values in binary (base 2) format. Values like `0.1` $= \frac{1}{10}$ are repeating decimals in base 2. (Just like $\frac{1}{3}$ is a repeating decimal in base 10.)

See https://docs.python.org/3.6/tutorial/floatingpoint.html for more information.

In [5]:
```
print(0.1 + 0.2)
print(0.1 + 0.2 - 0.3)
```

```
0.30000000000000004
5.551115123125783e-17
```

In [6]:
```
print(f"{0.1:.55f}") # this says "format 0.1 as a 55 decimal-place float"
print(f"{0.2:.55f}")
print(f"{0.3:.55f}")
```

```
0.1000000000000000055511151231257827021181583404541015625
0.2000000000000000111022302462515654042363166809082031250
0.2999999999999999888977697537484345957636833190917968750
```

Catalan Numbers

The Catalan numbers C_n are a sequence of integers 1, 1, 2, 5, 14, 42, 132... that play an important role in quantum mechanics and the theory of disordered systems. (They were central to Eugene Wigner's proof of the so-called semicircle law.) They are given by

$$C_0 = 1, \qquad C_{n+1} = \frac{4n+2}{n+2} C_n \,.$$

Let us create a numpy array of all Catalan numbers less than or equal to one billion. Note that the Catalan numbers are all integers.

In [7]:
```
# create an array of size 1, with first element C0
C0 = 1
catalan_1e9 = np.array([C0])
```

In Python, arrays start at 0. Notice what happens if we print the 0th element versus the 1st element of the array

In [8]:
```
catalan_1e9[0]
```

Out[8]:
```
1
```

In [9]:
```
catalan_1e9[1]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_26460\245894649.py in <module>
----> 1 catalan_1e9[1]

IndexError: index 1 is out of bounds for axis 0 with size 1
```

Recall the rest of the numbers are given by

$$C_0 = 1, \qquad C_{n+1} = \frac{4n+2}{n+2} C_n \,.$$

In [10]:
```
n = 0
while n>=0:  # while loop, continues until stopped
    Cn1 = int((4*n+2)/(n+2)*catalan_1e9[n])
    if Cn1<1e9:
        catalan_1e9 = np.append(catalan_1e9,Cn1)  # add the number to our array
        n += 1  # increase n by 1
    else:
        break
```

In [11]:
```
# Let us print the Catalan numbers

for Cn in catalan_1e9:
    print(Cn)
```

```
1
1
2
5
14
42
132
429
1430
4862
16796
58786
208012
742900
2674440
9694845
35357670
129644790
477638700
```

In [12]:
```
# check number of elements in our array
print(catalan_1e9.shape)
```

```
(19,)
```

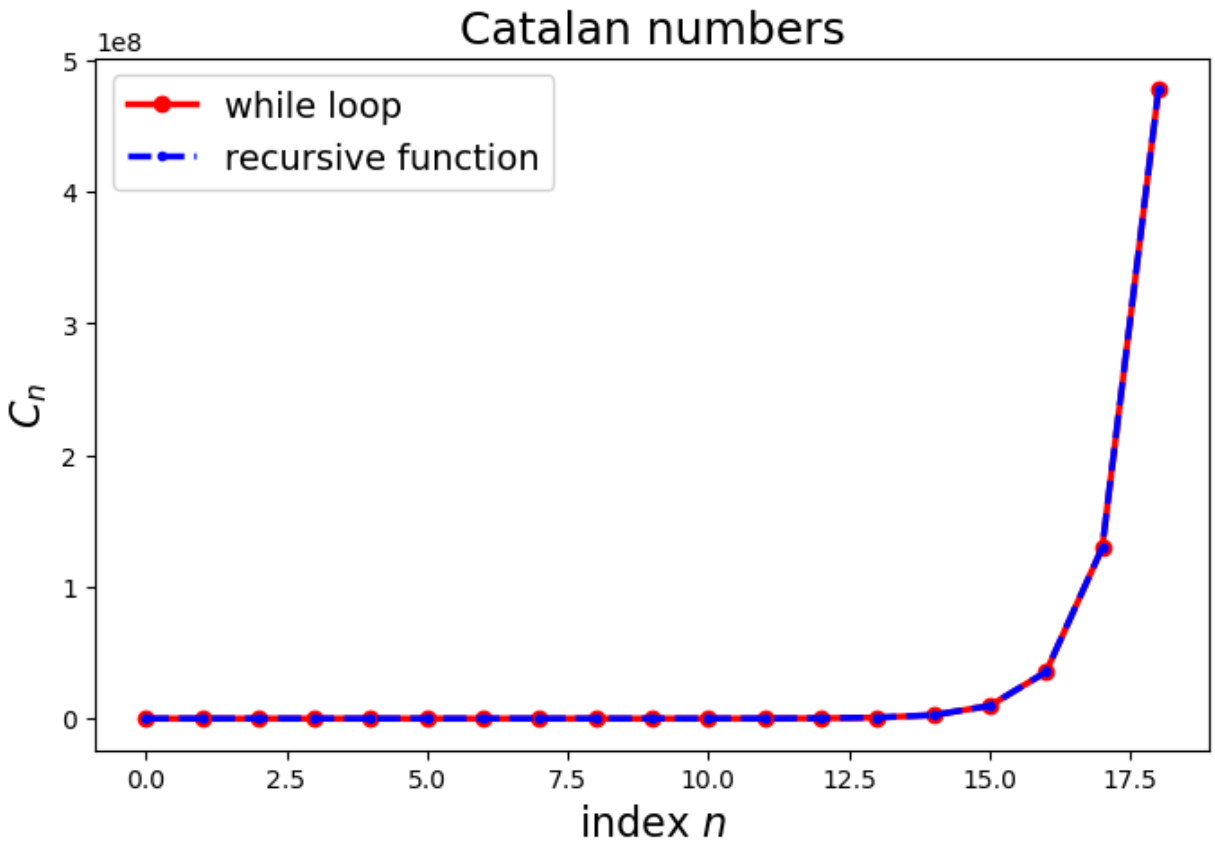
Alternatively, we can write a *function* to compute the next number given the current Catalan number, and use *recursion* to get the Catalan number of index n

In [13]:
```
def catalan(n):
    if n==0:
        return 1
    else:
        return int((4*n-2)/(n+1)*catalan(n-1))  # function calls itself
```

Let us plot the Catalan numbers using our function and confirm they match our prior code.

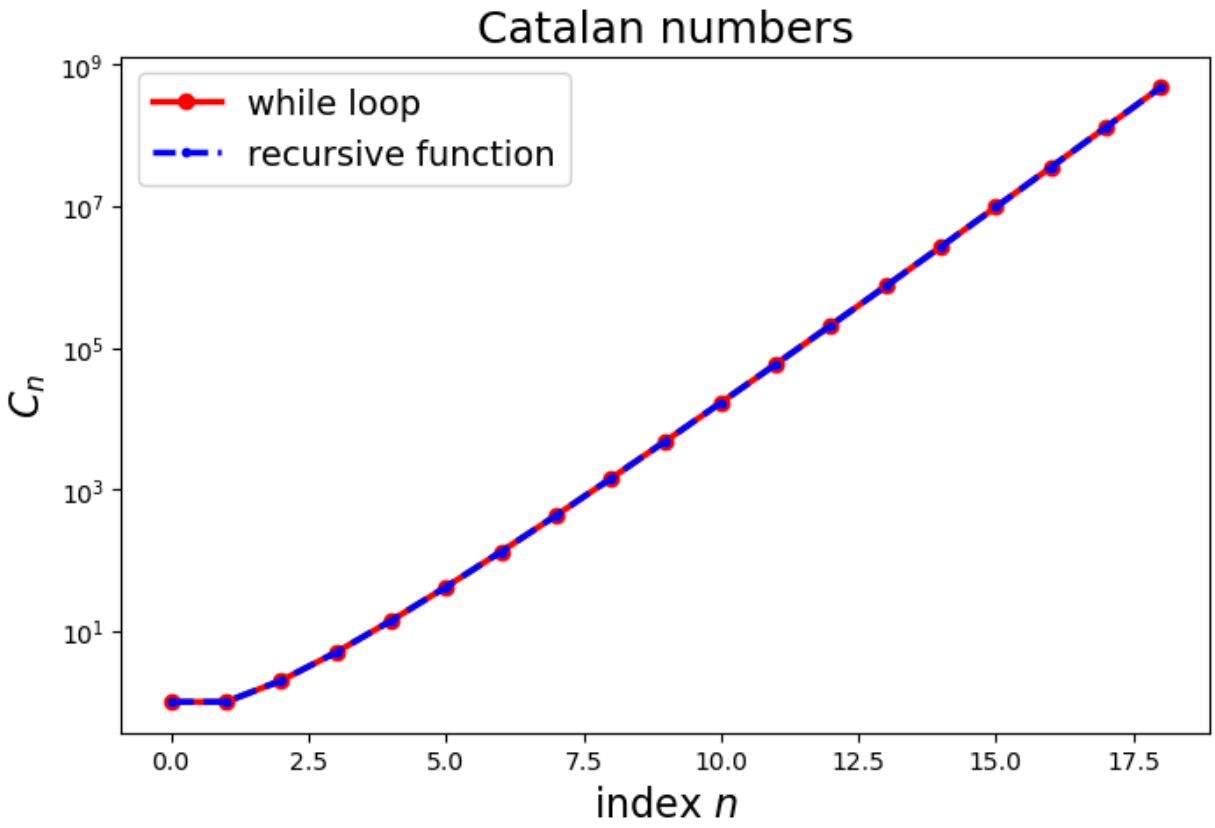
For matplotlib, there are a number of options when it comes to color, marker type, and line style. E.g., 'b' for blue, 's' for square marker, ':' for a dotted line syle

In [14]:
```
# the indices we want to plot
my_n_array = np.array(range(0,19,1))
# range gives you numbers from 0 to 19 in increments of 1, not including the final num

# we need to vectorize our function, because currently the inputs/outputs are floats
catalan_vectorized = np.vectorize(catalan)

# make a figure, plot the data, label axes, title plot
fig = plt.figure(figsize=(8,5),dpi = 100)
plt.plot(my_n_array,catalan_1e9,'ro-',linewidth=2.5,label="while loop")
plt.plot(my_n_array,catalan_vectorized(my_n_array),'b.--',linewidth=2.5,label="recursi
plt.xlabel('index $n$',fontsize=16)
plt.ylabel('$C_n$',fontsize=16)
plt.title('Catalan numbers',fontsize=18)
plt.legend(fontsize=14)
plt.show()  # crucial!
```



Notice how it is difficult to see the differences in numbers for small indices n. *Data visualization* is an important part of science and industry. Let's change the y-axis scaling:

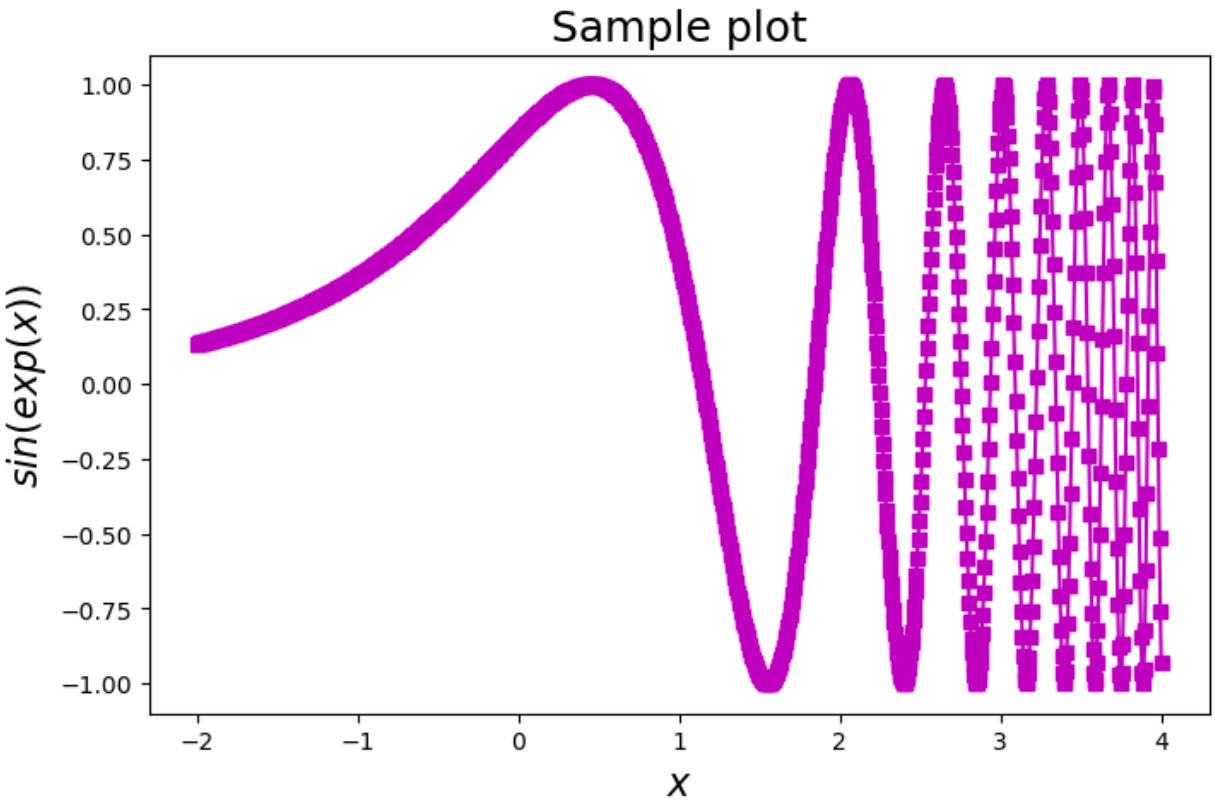
In [15]:
```
# make a figure, plot the data, label axes, title plot
fig = plt.figure(figsize=(8,5),dpi = 100)
plt.semilogy(my_n_array,catalan_1e9,'ro-',linewidth=2.5,label="while loop")
plt.semilogy(my_n_array,catalan_vectorized(my_n_array),'b.--',linewidth=2.5,label="rec
plt.xlabel('index $n$',fontsize=16)
plt.ylabel('$C_n$',fontsize=16)
plt.title('Catalan numbers',fontsize=18)
plt.legend(fontsize=14)

plt.show()  # crucial!
```



Math in numpy

We can use numpy functions to calculate things like $\sin$, $\exp$, etc.

In [16]:
```
# Let's plot sin(exp(x)) for x between -2 and 4

x = np.linspace(-2,4,1000);
y = np.sin(np.exp(x));

fig = plt.figure(figsize=(8,5),dpi = 100)
plt.plot(x,y,'ms-')
plt.xlabel('$x$',fontsize=16)
plt.ylabel('$sin(exp(x))$',fontsize=16)
plt.title('Sample plot',fontsize=18)
plt.show()
```



Saving figures

Let's save our figure as a pdf file

In [17]:
```
fig.savefig("FigExample.pdf",bbox_inches='tight')
```

A couple things to keep in mind

1) **Code readability matters**: Other people (and future you) should be able to read and use your scripts. Use good variable names, comment your code, etc.

2) **Test and debug your code**: If you come to office hours with an issue in your code, show what you have tested and checked throughout your script

3) **Write functions** and **use arrays**

4) **Make use of trusted libraries**: Other than when you are asked to write an implementation yourself (e.g., LU factorization to solve $Ax = b$ for x), use the wealth of widely employed and tested libraries like numpy for your code.

For more complex assignments, it's a good idea to click on

> **Kernel→Restart & Run All**

just to make sure everything runs cleanly before submitting your work.

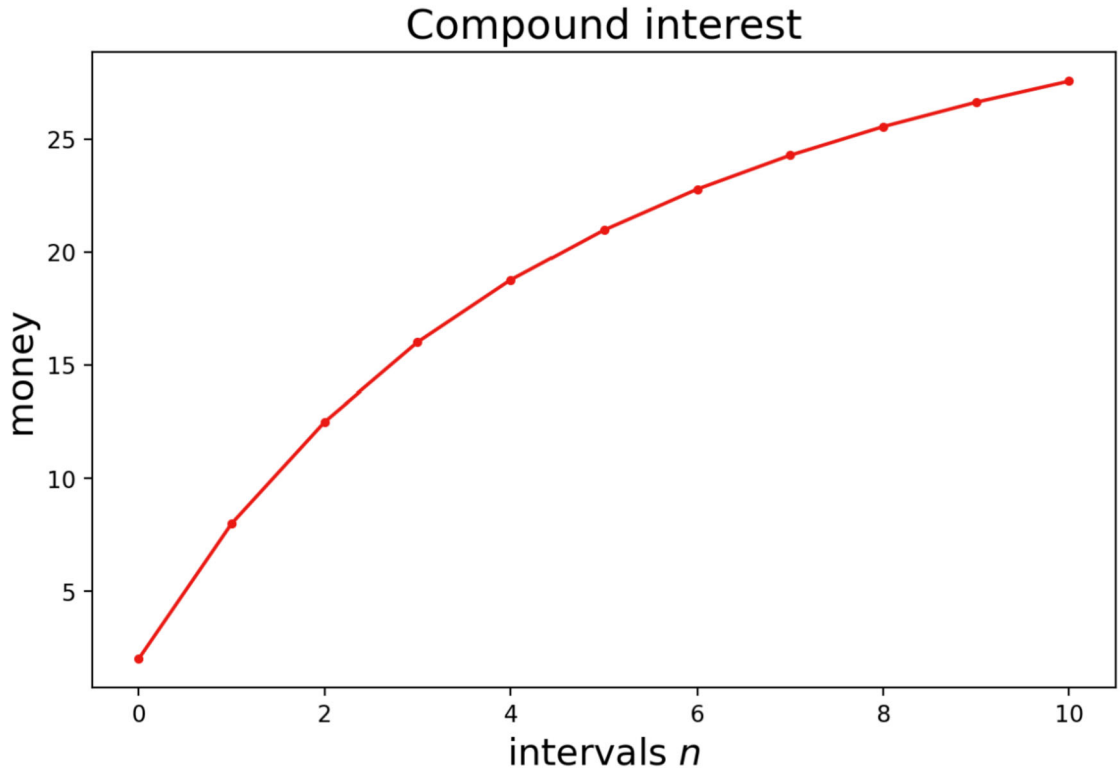
Try to do the following:

Write a function to compute compound interest, given a yearly interest rate $0 < r < 1$ compounded over n intervals on an amount of money C:

$$f(C, r, n) = C\left(1 + \frac{r}{n}\right)^n$$

Let $C = 2.0$ and $r = 3$ (so not < 1, but just as a fun check). Compute the interest for $n = 1, 2, 3, \ldots, 10$ intervals and plot the result.

The plot should look something like:



Exercise:

Newton's method in one dimension is given by:

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

Write a function and use recursion to find the root of $f(x) = -2 + x^3 + x + 1$ with accuracy $\epsilon = 1e-4$, given some initial condition. Plot the number of iterations taken to find the root depending on the initial condition for $x \in [-2, 2]$.

Is there anything notable about your result? Plot the function $f(x)$. Where might Newton's method have problems?

In [ ]: