PHZ4151C: Exercise 2

Computational Physics Lab
Due February 1

For each problem you will write one or more python programs. These programs should follow the Python 2.7.x coding and formatting conventions outlined for our course. You must hand in copies of the programs and outputs as prescribed in each problem.

In addition, you must submit all Python programs as an archive tgz file via email to phz4151c@hadron.physics.fsu.edu. Place copies of only your Python programs in a directory called <last_name>-exercise2/ where <last_name> is your last name. Below are the commands for creating, checking, and submitting archive files.

Creating archive file:

```
tar  -zcvf  last_name-exercise2.tgz   last_name-exercise2/
```

Checking archive contents:

```
tar -ztvf   last_name-exercise2.tgz
```

Submitting via email:

```
mail -s "exercise 2"  -c <your-email>  -a last_name-exercise2.tgz  phz4151c@hadron.physics.fsu.edu
```

Note: the mail command expects a message to be entered before sending the email. After entering the full "mail" command line one must end the message with either a "." on a new line then the [return] key or use the key combination [control-d] on a new line.

1. 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 defined by

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

Write a program that prints, in increasing order, all Catalan numbers less than or equal to ten billion. (do not use recursive methods)

For full credit turn in a copy of your final program and a copy of the output of your program.

2. The Madelung constant: In condensed matter physics the Madelung constant gives the total electric potential felt by an atom in a solid. It depends on the charges on the other atoms nearby and their locations. Consider for instance solid sodium chloride—table salt. The sodium chloride crystal has atoms arranged on a cubic lattice, but with alternating sodium and chlorine atoms, the sodium ones having a single positive charge $+e$ and the chlorine ones a single negative charge $-e$, where e is the charge on the electron. If we label each position on the lattice by three integer coordinates (i, j, k), then the sodium atoms fall at positions where $i + j + k$ is even, and the chlorine atoms at positions where $i + j + k$ is odd.

Consider a sodium atom at the origin, $i = j = k = 0$, and let us calculate the Madelung constant. If the spacing of atoms on the lattice is a, then the distance from the origin to the atom at position (i, j, k) is

$$\sqrt{(ia)^2+(ja)^2+(ka)^2} = a\sqrt{i^2+j^2+k^2},$$

and the potential at the origin created by such an atom is

$$V(i,j,k) = \pm \frac{e}{4\pi \epsilon_0 a \sqrt{i^2+j^2+k^2}},$$

with ε_0 being the permittivity of the vacuum and the sign of the expression depending on whether $i + j + k$ is even or odd. The total potential felt by the sodium atom is then the sum of this quantity over all other atoms. Let us assume a cubic box around the sodium at the origin, with L atoms in all directions. Then

$$V_{total} = \sum_{\substack{i,j,k=-L \\ \text{not } i=j=k=0}}^{L} V(i,j,k) = \frac{e}{4\pi \epsilon_0 a} M,$$

where M is the Madelung constant, at least approximately—technically the Madelung constant is the value of M when L → ∞, but one can get a good approximation just by using a large value of L.

Write a program to calculate and print the Madelung constant for sodium chloride. Use as large a value of L as you can, while still having your program run in reasonable time—say in a minute or less.

For full credit turn in a copy of your program, the answers you calculated using it, and a reasonable estimate of the program run time.

3. Binomial coefficients: The binomial coefficient $\binom{n}{k}$ is an integer equal to

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n\times(n-1)\times(n-2)\times\ldots\times(n-k+1)}{1\times2\times\ldots\times k}$$

when $k \geq 1$, or $\binom{n}{0} = 1$ when $k = 0$.

a) Using this form for the binomial coefficient, write a Python user-defined function `binomial(n,k)` that calculates the binomial coefficient for given n and k. Make sure your function returns the answer in the form of an integer (not a float) and gives the correct value of 1 for the case where $k = 0$.

b) Using your function write a program to print out the first 20 lines of "Pascal's triangle." The nth line of Pascal's triangle contains $n + 1$ numbers, which are the coefficients $\binom{n}{0}$, $\binom{n}{1}$, and so on up to $\binom{n}{n}$. Thus the first few lines are

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

c) The probability that an unbiased coin, tossed n times, will come up heads k times is $\binom{n}{k}/2^n$. Write a program to calculate (a) the total probability that a coin tossed $\boldsymbol{n}$ times comes up heads exactly $\boldsymbol{k}$ times, and (b) the probability that it comes up heads $\boldsymbol{k}$ or more times. Test your program with input values $\boldsymbol{n} = 137$ and $\boldsymbol{k} = 53$.

For full credit turn in copies of your two programs, a copy of the program output for part (b), and the answers you calculated for part (c).

4. **Recursion:** A useful feature of user-defined functions is *recursion*, the ability of a function to call itself. For example, consider the following definition of the factorial $n!$ of a positive integer n:

$$n! = \begin{Bmatrix} 1 \\ n\times(n-1)! \end{Bmatrix} \quad \begin{matrix} \text{if } n = 1, \\ \text{if } n > 1. \end{matrix}$$

This constitutes a complete definition of the factorial which allows us to calculate the value of $n!$ for any positive integer. We can employ this definition directly to create a Python function for factorials, like this:

```
def factorial(n):
     if n==1:
         return 1
     else:
         return n*factorial(n-1)
```

Note how, if n is not equal to 1, the function *calls itself* to calculate the factorial of $n - 1$. This is recursion. If we now say "`print(factorial(5))`" the computer will correctly print the answer 120.

(a) We encountered the Catalan numbers Cn previously in problem 1 above. With just a little rearrangement, the definition given there can be rewritten in the form

$$C_n = \begin{Bmatrix} 1 \\ \dfrac{4\text{n}-2}{n+1} C_{n-1} \end{Bmatrix} \quad \begin{matrix} \text{if } n = 0, \\ \text{if } n > 0. \end{matrix}$$

Write a Python function, using recursion, that calculates C_n. Use your function to calculate and print C_{110}.

(b) Euclid showed that the greatest common divisor $g(m,n)$ of two nonnegative integers m and n satisfies

$$g(m,n) = \begin{Bmatrix} m \\ g(n, m \mathit{mod}\, n) \end{Bmatrix} \quad \begin{matrix} \text{if } n = 0, \\ \text{if } n > 0. \end{matrix}$$

Write a Python function `g(m,n)` that employs recursion to calculate the greatest common divisor of m and n using this formula. Use your function to calculate and print the greatest common divisor of 108 and 180.

For full credit turn in copies of your two programs and the answers you calculated using them.