Tarea-Práctica 2.
Programación para la física computacional
Fundamentos de programación

Fecha de entrega: **4 de septiembre de 2023**

1. **Números de Catalan:**

   Los números de Catalan C_n son una secuencia de números enteros 1, 1, 2, 5, 14, 42, 132 que juegan un papel importante en la mecánica cuántica y la teoría de los sistemas desordenados. (Eran fundamentales para la prueba de Eugene Wigner de la llamada ley del semicírculo). Se definen por:

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

   Escribe un programa que imprima en orden creciente todos los números de Catalan C_n menores o iguales a $1x10^9$ (mil millones).

2. **La constante de Madelung**

   En física de la materia condensada, la *constante de Madelung* da el potencial eléctrico total que siente un átomo en un sólido; y depende de las cargas de los otros átomos cercanos y de sus ubicaciones.

   Por ejemplo, el cristal de cloruro de sodio sólido (la sal de mesa), tiene átomos dispuestos en una red cúbica, con átomos de sodio y cloro alternados, teniendo los de sodio una carga positiva $+e$ y los de cloro una negativa $-e$, (donde e es la carga del electrón). Si etiquetamos cada posición en la red con tres coordenadas enteras (i, j, k), entonces los átomos de sodio caen en posiciones donde $i + j + k$ es par, y los átomos de cloro en posiciones donde $i + j + k$ es impar.

   Consideremos un átomo de sodio en el origen, i.e. $i = j = k = 0$, y calculemos la *constante de Madelung*. Si el espaciado de los átomos en la red es a, entonces la distancia desde el origen al átomo en la posición (i, j, k) es:

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

   y el potencial en el origen creado por tal átomo es:

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

siendo ϵ_0 la permitividad del vacío y el signo de la expresión se toma dependiendo de si $i+j+k$ es par o impar. Así entonces, el potencial total que siente el átomo de sodio es la suma de esta cantidad sobre todos los demás átomos. Supongamos una caja cúbica alrededor del átomo de sodio en el origen, con L átomos en todas las direcciones, entonces:

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L \\ i,j,k\neq 0}}^{L} V(i,j,k) = \frac{e}{4\pi\epsilon_0 a} M,$$

donde M es la constante de Madelung (al menos aproximadamente).

Técnicamente, la constante de Madelung es el valor de M cuando $L \to \infty$, pero se puede obtener una buena aproximación simplemente usando un valor grande de L.

Escribe un programa para calcular e imprimir la *constante de Madelung* para el cloruro de sodio. Utiliza un valor de L tan grande como puedas, sin dejar que tu programa se ejecutar en un tiempo razonable (un minuto o menos).

3. **Coeficientes binomiales**

El coeficiente binomial $\binom{n}{k}$ es un número entero igual a:

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

donde $k \geq 1$, o bien $\binom{n}{0} = 1$ cuando $k = 0$.

(a) Utiliza esta formula para escribir una función llamada `binomial(n,k)` (o como tu quieras) que calcule el coeficiente binomial para un n y k dados. Asegúrate de que tu función devuelva la respuesta en forma de un número entero (no flotante) y proporcione el valor correcto de 1 para el caso en que $k = 0$.

(b) Usando tu función, escribe un programa que imprima las primeras 20 líneas del "*triángulo de Pascal*". La $n$ésima línea del triángulo de Pascal contiene $n+1$ números, que son los coeficientes $\binom{n}{0}$, $\binom{n}{1}$, y así sucesivamente hasta $\binom{n}{n}$. De tal manera que las primeras líneas son:

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

(c) La probabilidad de que para una moneda no sesgada, lanzada n veces, salga águila k veces es:

$$p(k|n) = \frac{\binom{n}{k}}{2^n}$$

Escribe un programa para calcular:

i. la probabilidad total de que una moneda lanzada 100 veces, salga águila exactamente 60 veces y

ii. la probabilidad de que salga águila 60 veces o más.

4. **Números primos**

Es posible escribir un programa bastante rápido para calcular números primos siguiendo las siguientes observaciones:

(a) Un número n es primo, si no tiene factores primos menores que n. Por lo tanto, sólo necesitamos comprobar si es divisible por otros números primos.

(b) Si un número n no es primo y tiene un factor r, entonces $n = rs$, donde s también es un factor. Si $r \geq \sqrt{n}$ entonces $n = rs \geq \sqrt{n}s$, lo que implica que $s \leq \sqrt{n}$. Es decir, cualquier **no primo** debe tener factores, y por lo tanto también factores primos, menores o iguales a $\sqrt{n}$. Por lo tanto, para determinar si un número es primo, tenemos que verificar sus factores primos solo hasta $\sqrt{n}$; incluso si no hay ninguno, entonces el número es primo.

(c) Si encontramos un **factor primo menor que $\sqrt{n}$ entonces sabemos que el número no es primo** y, por lo tanto, no hay necesidad de realizar más comprobaciones; podemos abandonar este número y pasar al siguiente.

Escribe un programa que encuentre todos los números primos hasta $10\,000$ **(diez mil)**. Crea una lista para almacenar los números primos, que comience con solo un número primo (el 2). Luego, para cada número n del 3 al $10\,000$, verifica si el número es divisible por alguno de los números primos de la lista hasta $\sqrt{n}$. En cuanto encuentres un factor primo, puedes dejar de comprobar el resto de números (pues ya sabrías que n no es primo). Si no encuentras factores primos iguales o menores a $\sqrt{n}$, entonces n es primo y habría que agregarlo a la lista. Puedes imprimir la lista completa al final del programa, o bien, ir imprimiendo los números individualmente a medida que los vayas encontrando.

5. **Recursión**

Una característica muy útil de las funciones que podemos definir es la ***recursividad***, es decir, la capacidad de una función de llamarse a sí misma. Por ejemplo, considere la siguiente definición del factorial $n!$ de un entero positivo n:

$$n! = \begin{cases} 1 & \text{si } n = 1, \\ n \times (n-1)! & \text{si } n > 1. \end{cases}$$

Esto constituye una definición completa del factorial que nos permite calcular el valor de $n!$ para cualquier entero positivo. Podemos emplear esta definición directamente para crear una función para calcular factoriales, como esta (en Python):

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

Nota cómo si n no es igual a 1, la función se llama a sí misma para calcular el factorial de $n-1$, de tal manera que al pedir "`print(factorial(5))`" el interprete de Python imprimirá correctamente la respuesta 120.

(a) Anteriormente en el ejercicio 1, encontramos los números de Catalan C_n. Con una pequeña reordenación, la definición de C_n se puede reescribir de la forma:

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

Escribe una función, usando recursividad, que calcule C_n. Utiliza tu función para calcular e imprimir C_{100}.

(b) Euclides demostró que el máximo común divisor $g(m,n)$ de dos enteros no negativos m y n satisface que:

$$g(m,n) = \begin{cases} m & \text{si } n = 0, \\ g(n, m \bmod n) & \text{si } n > 0. \end{cases}$$

Usando esta fórmula escribe una función `g(m,n)` que emplee recursividad para calcular el máximo común divisor de m y n. Usa tu función para calcular e imprimir el máximo común divisor de 108 y 192.

Comparando el cálculo de los números de Catalan del inciso 5a, con el del Ejercicio 1, vemos que es posible hacer el cálculo de dos maneras, ya sea directamente o usando *recursividad*.

En la mayoría de los casos, si una cantidad se puede calcular ***sin*** **recursividad,** será más rápido y normalmente se recomienda hacerlo así, si es posible. Sin embargo, hay algunos cálculos que son esencialmente imposibles (o al menos mucho más difíciles) sin recursividad.