03MIAR - Retos Actividades Guiadas

Nombre: Esmarlin Julissa Moreno Nivar
Github: https://github.com/julissrock/03MIAR-Algoritmos-de-Optimizacion/blob/main/AG_Retos.ipynb

In [6]:
```python
#Importar bibliotecas

import math
import numpy as np
```

Reto 1

Criba de Eratóstenes

El siguiente programa implementa el algoritmo de la Criba de Eratóstenes para encontrar números primos.

La Criba de Eratóstenes es un algoritmo eficiente que encuentra todos los números primos menores o iguales a un número dado n. Utiliza una lista de booleanos para marcar los números no primos y elimina múltiplos de números primos en el proceso. El resultado es una lista de números primos que se muestra al final.

In [3]:
```python
"""Definimos la función que encontrará todos los números primos menores o
iguales a n utilizando el algoritmo de la Criba de Eratóstenes."""

def criba_eratostenes(n): # n es el número hasta el que se encontraran los primos

  primos = [True] * (n + 1) # Inicializamos una lista de booleanos de tamaño n+1
  primos[0] = primos[1] = False # 0 y 1 no son primos

  for i in range(2, int(math.sqrt(n)) + 1): # Recorremos la lista hasta la raíz cuadrada de n
    if primos[i]:
      for j in range(i * i, n + 1, i):
        primos[j] = False

  return [i for i in range(2, n + 1) if primos[i]] # Retornamos los números primos hasta n

def main(): # Función main que llama a criba_eratostenes(n) y muestra el resultado
  n = 100 #Otra propuesta seria: n = int(input("Ingrese un número: "))
  primos = criba_eratostenes(n)
  print("Los números primos menores o iguales a {} son:".format(n)) # Mostramos el resultado
  print(primos) # Mostramos el resultado

if __name__ == "__main__":
  main() # Llamamos a la función principal() si el script se ejecuta directamente
```

```
Los números primos menores o iguales a 100 son:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Reto 2

Resolver el problema de las 8-Reinas.

El siguiente programa implementa la resolución del problema de las 8 reinas utilizando el algoritmo de backtracking. El objetivo es encontrar todas las posibles configuraciones de 8 reinas en un tablero de ajedrez de 8x8, donde ninguna reina amenaza a otra (ninguna reina comparte la misma fila, columna o diagonal con otra reina).

El algoritmo de backtracking utilizado busca exhaustivamente todas las posibles configuraciones de 8 reinas en el tablero, descartando aquellas configuraciones que violen las reglas del juego. Al final, se muestra el número total de soluciones encontradas.

In [11]:
```python
# Resolución del problema de las 8 reinas utilizando el algoritmo de backtracking.

def reinas(n): # n es el número de reinas que se quieren colocar en el tablero
    soluciones = [] # Inicializamos una lista vacía donde se guardarán las soluciones
```

```
    def backtracking(tablero, fila): # Función que implementa el algoritmo de backtracking

        if fila == n: # Si fila es igual a n, significa que se han colocado todas las reinas en el tablero
            soluciones.append(tablero) # Guardamos la solución en la lista soluciones
            return

        for columna in range(n): # Iteramos sobre las columnas del tablero
            if es_valida(tablero, fila, columna):
                tablero[fila] = columna
                backtracking(tablero, fila + 1)

    tablero = [-1] * n # Inicializamos el tablero con -1 en cada posición
    backtracking(tablero, 0) # Llamamos a la función retroceso(tablero, fila) con fila = 0

    return soluciones # Devolvemos la lista de soluciones



def es_valida(tablero, fila, columna): # Función que verifica si la reina es válida

    for i in range(fila):  # Iteramos y verificamos si hay reina en la columna o diagonal
        if tablero[i] == columna or abs(tablero[i] - columna) == fila - i:
            return False

    return True



def main(): # Función main que llama a reinas(n) y muestra los resultados según n

    soluciones = reinas(8) # Llamamos a reinas(n) y guardamos el resultado en soluciones
    print("El número de soluciones es: {}".format(len(soluciones)))


if __name__ == "__main__":
  main() # Llamamos a la función principal() si el script se ejecuta directamente
```

```
El número de soluciones es: 92
```

Reto 3

Resolver el problema de las Torres de Hanoi para un número arbitrario de fichas.

3.1 Función recursiva: El siguiente programa implementa un algoritmo que utiliza una función recursiva que resuelve el problema de las Torres de Hanoi para un número albritario de fichas utilizando el enfoque Divide y Venceras.

El algoritmo se implementa de manera recursiva, dividiendo el problema en subproblemas más pequeños hasta llegar al caso base (una ficha o disco). Luego, se resuelven los subproblemas de forma ascendente, moviendo las fichas en la dirección correcta hasta completar la tarea de mover todas las fichas de la torre de origen a la torre de destino.

In [5]:
```
def torres_de_hanoi_recursivo(n, origen, auxiliar, destino):
    # Función del algoritmo de las torres de Hanoi

    if n == 1:
        # Si solo hay una ficha, se mueve directamente a la torre de destino
        print("Mover ficha 1 desde {} hasta {}".format(origen, destino))
    else:
        # Mover n-1 fichas de la torre de origen a la torre auxiliar,
        # utilizando la torre de destino como torre auxiliar
        torres_de_hanoi_recursivo(n - 1, origen, destino, auxiliar)

        # Mover la ficha de la torre de origen a la torre de destino
        print("Mover ficha {} desde {} hasta {}".format(n, origen, destino))

        # Mover los n-1 fichas de la torre auxiliar a la torre de destino, #
        utilizando la torre de origen como torre auxiliar
        torres_de_hanoi_recursivo(n - 1, auxiliar, origen, destino)


def main():  # Main inicializa el número de fichas y llama a torres_de_hanoi_recursivo.
    n = 4  # Número de fichas
    torres_de_hanoi_recursivo(n, "A", "B", "C") # Llamamos a la función recursiva
```

```
if __name__ == "__main__":
    main()  # Ejecución de la función principal si el archivo se ejecuta directamente
```

```
Mover ficha 1 desde A hasta B
Mover ficha 2 desde A hasta C
Mover ficha 1 desde B hasta C
Mover ficha 3 desde A hasta B
Mover ficha 1 desde C hasta A
Mover ficha 2 desde C hasta B
Mover ficha 1 desde A hasta B
Mover ficha 4 desde A hasta C
Mover ficha 1 desde B hasta C
Mover ficha 2 desde B hasta A
Mover ficha 1 desde C hasta A
Mover ficha 3 desde B hasta C
Mover ficha 1 desde A hasta B
Mover ficha 2 desde A hasta C
Mover ficha 1 desde B hasta C
```

3.2 Otra propuesta utilizando un Algoritmo Iterativo para resolver el problema de las Torres de Hanoi para un número arbitrario de fichas:

El algoritmo de las Torres de Hanoi se puede resolver de diferentes maneras, y el enfoque de "divide y vencerás" es el más común y eficiente. Sin embargo, existen otras aproximaciones para resolver el problema de las Torres de Hanoi, aunque pueden ser menos eficientes en términos de tiempo y espacio.

Una de las alternativas es utilizar un enfoque iterativo o basado en bucles en lugar de la recursividad. En este enfoque, se utilizan estructuras de datos como pilas o colas para simular las operaciones de movimiento de discos entre las torres. A medida que se realizan los movimientos, se actualizan las pilas o colas para reflejar los cambios en la configuración de los discos. Este enfoque puede ser más fácil de entender y visualizar, pero tiende a ser menos eficiente en comparación con el enfoque de "divide y vencerás".

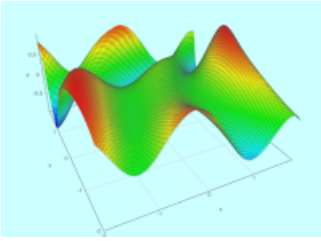
El algoritmo que mostramos a continuación se conoce como "Iterative Towers of Hanoi" (Torres de Hanoi iterativo). Este algoritmo utiliza una pila (stack) para realizar un enfoque iterativo, en cada iteración, se extrae un subproblema de la pila y se resuelve. Luego, se generan y agregan a la pila los subproblemas más pequeños. Este proceso se repite hasta que no queden subproblemas pendientes en la pila, es decir, hasta que se haya movido todas las fichas a la torre de destino.

In [13]:
```
def torres_de_hanoi_iterativo(n): # Función para torres de Hanoi de forma iterativa
    stack = [(n, 'A', 'B', 'C')] # Inicializamos una pila

    while stack:
        ficha, origen, aux, destino = stack.pop()
        # Mientras la pila no esté vacía sacamos el último elemento de la pila

        if ficha == 1:
            # Mover ficha directamente a la torre de destino
            print(f"Mover ficha 1 desde {origen} hasta {destino}")
        else:
            # Empujar los subproblemas a la pila
            stack.append((ficha - 1, aux, origen, destino))
            stack.append((1, origen, aux, destino))
            stack.append((ficha - 1, origen, destino, aux))

# Ejemplo de uso
n = 4  # Número de fichas
torres_de_hanoi_iterativo(n)
```

```
Mover ficha 1 desde A hasta B
Mover ficha 1 desde A hasta C
Mover ficha 1 desde B hasta C
Mover ficha 1 desde A hasta B
Mover ficha 1 desde C hasta A
Mover ficha 1 desde C hasta B
Mover ficha 1 desde A hasta B
Mover ficha 1 desde A hasta C
Mover ficha 1 desde B hasta C
Mover ficha 1 desde B hasta A
Mover ficha 1 desde C hasta A
Mover ficha 1 desde B hasta C
Mover ficha 1 desde A hasta B
Mover ficha 1 desde A hasta C
Mover ficha 1 desde B hasta C
```

Reto 4.

Optimización de la función $f(x) = sin(1/2 * x^2 - 1/4 * y^2 + 3) * cos(2 * x + 1 - e^y)$ mediante el algoritmo por descenso del gradiente.



El algoritmo implementa el método de descenso del gradiente para optimizar la función
$f(x, y) = sin(0.5 * x^2 - 0.25 * y^2 + 3) * cos(2 * x + 1 - e^y)$
El objetivo es encontrar los valores de x & y que minimizan esta función.

El descenso del gradiente es un algoritmo de optimización que se basa en el cálculo del gradiente de una función en un punto dado.

Este algoritmo busca encontrar un mínimo local de la función f(x, y) mediante la actualización iterativa de los valores de x y y en dirección opuesta al gradiente. Sin embargo, es importante tener en cuenta que el descenso del gradiente no garantiza encontrar el mínimo global, ya que puede quedar atrapado en mínimos locales.

In [16]:
```
"""
La función f(x, y) se define en la función f().
El gradiente de la función f(x, y) se define en la función grad_f().
El algoritmo de descenso del gradiente se define en la función gradient_descent().
Los valores de X y Y se inicializan en 0 y 0, respectivamente.
"""

def f(x, y): #Calculamos f(x, y) = sin(0.5 * x^2 - 0.25 * y^2 + 3) * cos(2*x + 1 - e^y)
    return math.sin(0.5 * x**2 - 0.25 * y**2 + 3) * math.cos(2 * x + 1 - math.exp(y))

def grad_f(x, y): #Calculamos el gradiente de f(x, y)
    return [
        math.cos(0.5 * x**2 - 0.25 * y**2 + 3) * (1 + x),
        -0.25 * math.sin(0.5 * x**2 - 0.25 * y**2 + 3) * math.cos(2 * x + 1 - math.exp(y)) + math.sin(2 * x
    ]

def gradient_descent(x, y, learning_rate, iterations): #Calculamos el descenso del gradiente
    for _ in range(iterations):
        gradient = grad_f(x, y)
        x -= learning_rate * gradient[0]
        y -= learning_rate * gradient[1]
    return x, y

x, y = gradient_descent(0, 0, 0.01, 1000)
print(x, y)
```

```
1.9070010264178119 0.650961780390601
```

Reto 5

Implementar el algoritmo del descenso por gradiente para el problema de la regresión lineal simple.

La regresión lineal es un modelo estadístico que busca establecer una relación lineal entre una variable dependiente y una o más variables independientes.

El algoritmo implementa el descenso de gradiente para una regresión lineal simple. A través de iteraciones, ajusta los coeficientes de la línea de regresión (pendiente y término de intersección) utilizando la tasa de aprendizaje y el gradiente

calculado a partir del error entre los valores reales y los valores predichos. El objetivo es minimizar el error y obtener una línea que se ajuste mejor a los datos de entrada. Al finalizar, devuelve los coeficientes del modelo de regresión lineal.

In [15]:
```
def descenso_gradiente(x, y, tasa_aprendizaje, iteraciones):
  """
  Realiza el descenso de gradiente para una regresión lineal simple.

  Argumentos:
    x: Los datos de entrada.
    y: Los datos de salida.
    tasa_aprendizaje: La tasa de aprendizaje.
    iteraciones: El número de iteraciones.

  Devuelve:
    Los coeficientes del modelo de regresión lineal.
  """

  m = 0
  b = 0

  for _ in range(iteraciones): # Iteramos para actualizar los valores de m y b
    error = y - (m * x + b)
    gradiente_m = np.sum(error * x)
    gradiente_b = np.sum(error)
    m -= tasa_aprendizaje * gradiente_m
    b -= tasa_aprendizaje * gradiente_b

  return m, b

def main(): # Función principal que inicializa los datos de entrada
            # y llama a la función descenso_gradiente

  x = np.array([1, 2, 3, 4, 5])
  y = np.array([2, 4, 6, 8, 10])
  m, b = descenso_gradiente(x, y, 0.01, 1000)
  print("Los coeficientes del modelo de regresión lineal son: m = {}, b = {}".format(m, b))

if __name__ == "__main__":
  main() # Llamamos a la función principal() si el script se ejecuta directamente
```

```
Los coeficientes del modelo de regresión lineal son: m = -1.2261899797249498e+202, b = -3.396349623642558e+
201
```