# AG1 - Actividad Guiada 1

Nombre: Alberto Rodriguez Arizaga

Actividad guiada de Algoritmos Optimización (AG1)

https://colab.research.google.com/drive/1tT6BBppj83w9r8RWccMekl9FN1OBQVMP?usp=sharing

https://github.com/brtln05/03MIAR---Algoritmos-de-Optimizacion

# Problema de las torres de Hanoi. Técnica de divide y vencerás.

```
#Torres de Hanoi - Divide y venceras
#################################################################################


#####################################################################
def Torres_Hanoi(N, desde, hasta):
  #N - Nº de fichas
  #desde - torre inicial
  #hasta - torre final
  if N==1 :
    print(f"Lleva la ficha desde {desde}  hasta {hasta}")


  else:
    Torres_Hanoi(N-1, desde, 6-desde-hasta)
    print(f"Lleva la ficha desde {desde}  hasta {hasta}")
    Torres_Hanoi(N-1, 6-desde-hasta,  hasta)

Torres_Hanoi(3, 1, 3)
#####################################################################
```

```
Lleva la ficha desde 1  hasta 3
Lleva la ficha desde 1  hasta 2
Lleva la ficha desde 3  hasta 2
Lleva la ficha desde 1  hasta 3
Lleva la ficha desde 2  hasta 1
Lleva la ficha desde 2  hasta 3
Lleva la ficha desde 1  hasta 3
```

# Problema del cambio de monedas. Técnica de algoritmo voraz.

✓ 0 s completado a las 19:19

```
  # Cantidad: Cantidad de dinero a cambiar
  # Sistema: Diferentes monedas que contiene nuestro sistema monetario

  solucion = {}
  sistema.sort(reverse=True)

  for tipo_moneda in sistema:

    cant_monedas = cantidad // tipo_moneda

    solucion[tipo_moneda] = cant_monedas

    cantidad -= cant_monedas * tipo_moneda

    if cantidad == 0:

      return solucion
  print("No es posible encontrar solucion")

cambio_monedas(15,[12, 5 ,2, 1  ])
```

```
{12: 1, 5: 0, 2: 1, 1: 1}
```

## Problema de las 4 reinas. Técnica de la vuelta atrás

```
#Verifica que en la solución parcial no hay amenzas entre reinas
def es_prometedora(solucion,etapa):

  #print(solucion)
  #Si la solución tiene dos valores iguales no es valida => Dos reinas en la misma fila
  for i in range(etapa+1):
    #print("El valor " + str(solucion[i]) + " está " +  str(solucion.count(solucion[i])) +
    if solucion.count(solucion[i]) > 1:
      return False

    #Verifica las diagonales
    for j in range(i+1, etapa +1 ):
      #print("Comprobando diagonal de " + str(i) + " y " + str(j))
      if abs(i-j) == abs(solucion[i]-solucion[j]) :
        return False
  return True
```

```
# Proceso principal de N-Reinas
def reinas(N, solucion=[],etapa=0):

  # Inicia el proceso una lista con posiciones vacías
  if len(solucion) == 0:
    solucion = [0 for i in range(N) ]

  for i in range(N):
    solucion[etapa] = i+1
    if es_prometedora(solucion, etapa):
      if etapa == N-1:
        print("Solución ",solucion)
      else:
        reinas(N, solucion, etapa+1)
    else:
      pass

  solucion[etapa] = 0

reinas(4,solucion=[],etapa=0)
```

```
 Solución  [2, 4, 1, 3]
 Solución  [3, 1, 4, 2]
```

```
# Dibuja la solución al problema  de N-Reinas
def escribe_solucion(S):
  n = len(S)
  for x in range(n):
    print("")
    for i in range(n):
      if S[i] == x+1:
        print(" X " , end="")
      else:
        print(" - ", end="")

escribe_solucion([2, 4, 1, 3])
```

```
 -  -  X  -
 X  -  -  -
 -  -  -  X
 -  X  -  -
```

# Problema: Encontrar los dos puntos más cercanos

## 1. Suponer en 1D, o sea, una lista de números. Calcular por fuerza bruta

Para calcularlo por fuerza bruta realizamos un bucle que recorra todos los elementos. Suponemos en este caso que la lista no esta ordenada.

```
lista_puntos_1d = [7,2,4,10,15,25,39,59,47,32,72,38]


def puntos_cercanos(puntos):

    # Inicializamos la función
    distancia_cercana = float('inf')
    puntos_cercanos = (None, None)
    # Construimos el bucle que recorra cada elemento
    # En cada j empezara comparando solo los puntos siguientes puesto que los
    # anteriores ya fueron comparadaos
    for i in range(len(puntos)):
        for j in range(i + 1, len(puntos)):

            distance = abs(puntos[i] - puntos[j])

            if distance < distancia_cercana:
                distancia_cercana = distance
                puntos_cercanos = (puntos[i], puntos[j])

    return puntos_cercanos


puntos_cercanos(lista_puntos_1d)
```

```
(39, 38)
```

## 2. Calcular la complejidad. ¿Se puede mejorar?

En este caso la complejidad sería de O(n^2) puesto que el bucle se recorrería como la suma (n−1)+(n−2)+ ... +1 lo cual suma (1/2)(n^2−n).

Esta complejidad puede mejorarse simplemente ordenando la lista de puntos pasando a ser de O(n) o usando un algoritmo de divide y vencerás como el que vemos en el punto siguiente.

## 3. Segundo intento. Aplicar Divide y Vencerás. Calcular la complejidad.

En este caso, el problema se va diviendo sucesivamente en mitades, hasta llegar al caso base donde se calcula la distancia y se construye hasta el caso completo. Este proceso tiene complejidad O(n log(n)). Su implementación se puede ver aquí abajo.

En este caso la implementación utiliza parte de una lista de puntos en 1D.

```
def distancia_1d(puntos):

    return abs(puntos[0]-(puntos[1]))

def puntos_cercanos(puntos):

    n = len(puntos)

    # Casos base

    if n <= 1:
        return [],float('inf')

    if n == 2:

        return puntos, distancia_1d(puntos)

    # Dividimos el conjunto de puntos en dos mitades

    mid = n // 2
    izquierda_lado = puntos[:mid]
    derecha_lado = puntos[mid:]

    # Comenzamos el proceso recursivo calculando en ambos lados

    punto_izqu, d_izquierda = puntos_cercanos(izquierda_lado)

    punto_der, d_derecha = puntos_cercanos(derecha_lado)



    distancia = min(d_izquierda, d_derecha)

    if distancia == d_derecha:
        punto_close = punto_der
    else:
        punto_close = punto_izqu

    # Calculamos los puntos cuya distancia al pivote es inferior a la distancia en cada la

    pivote = [punto for punto in puntos if abs(punto - puntos[mid]) < distancia]

    pivote.sort()

    m = len (pivote)

    # Ahora podemos calcular por fuerza bruta las distancias al pivote
    # y coger la mínima y los puntos asociados.
```

```
    for i in range(m - 1):
        for j in range(i + 1, min(i + 7, m)):
            distancia = min(distancia, abs(pivote[i] - pivote[j]))

            if distancia ==  abs(pivote[i] - pivote[j]):
                punto_close = [pivote[i], pivote[j]]



    return punto_close, distancia


puntos_cercanos(lista_puntos_1d)
```

```
([38, 39], 1)
```