

# Actividad Guiada 1 de Algoritmos de Optimización

Nombre: Roberto Merchán González

Github: <https://github.com/robertomergon/03MIAR-Algoritmos-de-Optimizacion>

## Ejercicio 1: Torres de Hanoi (Aplicando recursividad)

```
In [ ]: def Torres_Hanoi(N, desde, hasta):
    #N - Nº de fichas
    #desde - torre inicial
    #hasta - torre final
    if N==1 :
        print("Lleva la ficha " + str(N) + " desde " + str(desde) + " hasta " + str(hasta))
    else:
        Torres_Hanoi(N-1, desde, 6-desde-hasta)
        print("Lleva la ficha " + str(N) + " desde " + str(desde) + " hasta " + str(hasta))
        Torres_Hanoi(N-1, 6-desde-hasta, hasta)

Torres_Hanoi(4, 1, 3)
```

```
Lleva la ficha 1 desde 1 hasta 2
Lleva la ficha 2 desde 1 hasta 3
Lleva la ficha 1 desde 2 hasta 3
Lleva la ficha 3 desde 1 hasta 2
Lleva la ficha 1 desde 3 hasta 1
Lleva la ficha 2 desde 3 hasta 2
Lleva la ficha 1 desde 1 hasta 2
Lleva la ficha 4 desde 1 hasta 3
Lleva la ficha 1 desde 2 hasta 3
Lleva la ficha 2 desde 2 hasta 1
Lleva la ficha 1 desde 3 hasta 1
Lleva la ficha 3 desde 2 hasta 3
Lleva la ficha 1 desde 1 hasta 2
Lleva la ficha 2 desde 1 hasta 3
Lleva la ficha 1 desde 2 hasta 3
```

## Ejercicio 2: Cambio de monedas (Aplicando algoritmo voraz)

Se tomará como óptimo el algoritmo que minimice el número de monedas a devolver, es decir, el algoritmo que devuelverá las monedas de mayor valor posible en cada iteración.

```
In [ ]: SISTEMA = [50, 20, 10, 5, 1]
CANTIDAD = 78
SOLUCION = [1, 1, 0, 1, 3]
```

```

def cambio_monedas(CANTIDAD,SISTEMA):
#.....
    SOLUCION = [0]*len(SISTEMA)
    ValorAcumulado = 0

    for i,valor in enumerate(SISTEMA):
        monedas = (CANTIDAD-ValorAcumulado)//valor
        SOLUCION[i] = monedas
        ValorAcumulado = ValorAcumulado + monedas*valor

    if CANTIDAD == ValorAcumulado:
        return SOLUCION

    print("No es posible encontrar solución en la que se devuelva el cambio exacto")

cambios = cambio_monedas(CANTIDAD,SISTEMA)

print("Se han dado las siguientes monedas: \n")
for i in range(len(SISTEMA)):
    print(str(cambios[i]) + " monedas de " + str(SISTEMA[i]) + "€")

print("En total se han dado " + str(sum(cambios)) + " monedas")

```

Se han dado las siguientes monedas:

1 monedas de 50€  
 1 monedas de 20€  
 0 monedas de 10€  
 1 monedas de 5€  
 3 monedas de 1€  
 En total se han dado 6 monedas

**El algoritmo no siempre puede llegar a una solución, ya que si el sistema no contiene las monedas necesarias para devolver el cambio, el algoritmo no podrá llegar a una solución en la que se devuelva el cambio exacto.**

Por ejemplo:

```
In [ ]: SISTEMA = [25, 10, 5]
CANTIDAD = 33

cambio_monedas(CANTIDAD,SISTEMA)
```

No es posible encontrar solución en la que se devuelva el cambio exacto

**Otro problema que puede tener el algoritmo es cuando tenemos un número limitado de cada moneda y no podemos devolver el cambio exacto.**

## Ejercicio 3: Problema de las N-Reinas (Aplicando backtracking)

```
In [ ]: 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

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="")

def reinas(N, solucion=[], etapa=0):
    if len(solucion) == 0:           # [0,0,0...]
        solucion = [0 for i in range(N) ]
    for i in range(1, N+1):
        solucion[etapa] = i
        if es_prometedora(solucion, etapa):
            if etapa == N-1:
                #escribe_solucion(solucion)
                print(solucion)
            else:
                reinas(N, solucion, etapa+1)
        else:
            None

    solucion[etapa] = 0

reinas(8,solucion=[],etapa=0)

```

```
[1, 5, 8, 6, 3, 7, 2, 4]
[1, 6, 8, 3, 7, 4, 2, 5]
[1, 7, 4, 6, 8, 2, 5, 3]
[1, 7, 5, 8, 2, 4, 6, 3]
[2, 4, 6, 8, 3, 1, 7, 5]
[2, 5, 7, 1, 3, 8, 6, 4]
[2, 5, 7, 4, 1, 8, 6, 3]
[2, 6, 1, 7, 4, 8, 3, 5]
[2, 6, 8, 3, 1, 4, 7, 5]
[2, 7, 3, 6, 8, 5, 1, 4]
[2, 7, 5, 8, 1, 4, 6, 3]
[2, 8, 6, 1, 3, 5, 7, 4]
[3, 1, 7, 5, 8, 2, 4, 6]
[3, 5, 2, 8, 1, 7, 4, 6]
[3, 5, 2, 8, 6, 4, 7, 1]
[3, 5, 7, 1, 4, 2, 8, 6]
[3, 5, 8, 4, 1, 7, 2, 6]
[3, 6, 2, 5, 8, 1, 7, 4]
[3, 6, 2, 7, 1, 4, 8, 5]
[3, 6, 2, 7, 5, 1, 8, 4]
[3, 6, 4, 1, 8, 5, 7, 2]
[3, 6, 4, 2, 8, 5, 7, 1]
[3, 6, 8, 1, 4, 7, 5, 2]
[3, 6, 8, 1, 5, 7, 2, 4]
[3, 6, 8, 2, 4, 1, 7, 5]
[3, 7, 2, 8, 5, 1, 4, 6]
[3, 7, 2, 8, 6, 4, 1, 5]
[3, 8, 4, 7, 1, 6, 2, 5]
[4, 1, 5, 8, 2, 7, 3, 6]
[4, 1, 5, 8, 6, 3, 7, 2]
[4, 2, 5, 8, 6, 1, 3, 7]
[4, 2, 7, 3, 6, 8, 1, 5]
[4, 2, 7, 3, 6, 8, 5, 1]
[4, 2, 7, 5, 1, 8, 6, 3]
[4, 2, 8, 5, 7, 1, 3, 6]
[4, 2, 8, 6, 1, 3, 5, 7]
[4, 6, 1, 5, 2, 8, 3, 7]
[4, 6, 8, 2, 7, 1, 3, 5]
[4, 6, 8, 3, 1, 7, 5, 2]
[4, 7, 1, 8, 5, 2, 6, 3]
[4, 7, 3, 8, 2, 5, 1, 6]
[4, 7, 5, 2, 6, 1, 3, 8]
[4, 7, 5, 3, 1, 6, 8, 2]
[4, 8, 1, 3, 6, 2, 7, 5]
[4, 8, 1, 5, 7, 2, 6, 3]
[4, 8, 5, 3, 1, 7, 2, 6]
[5, 1, 4, 6, 8, 2, 7, 3]
[5, 1, 8, 4, 2, 7, 3, 6]
[5, 1, 8, 6, 3, 7, 2, 4]
[5, 2, 4, 6, 8, 3, 1, 7]
[5, 2, 4, 7, 3, 8, 6, 1]
[5, 2, 6, 1, 7, 4, 8, 3]
[5, 2, 8, 1, 4, 7, 3, 6]
[5, 3, 1, 6, 8, 2, 4, 7]
[5, 3, 1, 7, 2, 8, 6, 4]
[5, 3, 8, 4, 7, 1, 6, 2]
[5, 7, 1, 3, 8, 6, 4, 2]
[5, 7, 1, 4, 2, 8, 6, 3]
[5, 7, 2, 4, 8, 1, 3, 6]
[5, 7, 2, 6, 3, 1, 4, 8]
```

```
[5, 7, 2, 6, 3, 1, 8, 4]
[5, 7, 4, 1, 3, 8, 6, 2]
[5, 8, 4, 1, 3, 6, 2, 7]
[5, 8, 4, 1, 7, 2, 6, 3]
[6, 1, 5, 2, 8, 3, 7, 4]
[6, 2, 7, 1, 3, 5, 8, 4]
[6, 2, 7, 1, 4, 8, 5, 3]
[6, 3, 1, 7, 5, 8, 2, 4]
[6, 3, 1, 8, 4, 2, 7, 5]
[6, 3, 1, 8, 5, 2, 4, 7]
[6, 3, 5, 7, 1, 4, 2, 8]
[6, 3, 5, 8, 1, 4, 2, 7]
[6, 3, 7, 2, 4, 8, 1, 5]
[6, 3, 7, 2, 8, 5, 1, 4]
[6, 3, 7, 4, 1, 8, 2, 5]
[6, 4, 1, 5, 8, 2, 7, 3]
[6, 4, 2, 8, 5, 7, 1, 3]
[6, 4, 7, 1, 3, 5, 2, 8]
[6, 4, 7, 1, 8, 2, 5, 3]
[6, 8, 2, 4, 1, 7, 5, 3]
[7, 1, 3, 8, 6, 4, 2, 5]
[7, 2, 4, 1, 8, 5, 3, 6]
[7, 2, 6, 3, 1, 4, 8, 5]
[7, 3, 1, 6, 8, 5, 2, 4]
[7, 3, 8, 2, 5, 1, 6, 4]
[7, 4, 2, 5, 8, 1, 3, 6]
[7, 4, 2, 8, 6, 1, 3, 5]
[7, 5, 3, 1, 6, 8, 2, 4]
[8, 2, 4, 1, 7, 5, 3, 6]
[8, 2, 5, 3, 1, 7, 4, 6]
[8, 3, 1, 6, 2, 5, 7, 4]
[8, 4, 1, 3, 6, 2, 7, 5]
```

In [ ]: `escribe_solucion([6, 8, 2, 4, 1, 7, 5, 3])`

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

## Ejercicio 4: Viaje por el río (Aplicando programación dinámica)

In [ ]: `#Viaje por el río - Programación dinámica`

```
TARIFAS = [
[0,5,4,3,999,999,999],
[999,0,999,2,3,999,11],
[999,999, 0,1,999,4,10],
[999,999,999, 0,5,6,9],
[999,999, 999,999,0,999,4],
[999,999, 999,999,999,0,3],
[999,999,999,999,999,999,0]
]
```

```

#999 se puede sustituir por float("inf")

#Calculo de la matriz de PRECIOS y RUTAS
#####
def Precios(TARIFAS):
#####
    #Total de Nodos
    N = len(TARIFAS[0])

    #Inicialización de la tabla de precios
    PRECIOS = [ [9999]*N for i in [9999]*N]
    RUTA = [ [""]*N for i in [""]*N]

    for i in range(0,N-1):
        RUTA[i][i] = i           #Para ir de i a i se "pasa por i"
        PRECIOS[i][i] = 0         #Para ir de i a i se paga 0
        for j in range(i+1, N):
            MIN = TARIFAS[i][j]
            RUTA[i][j] = i

            for k in range(i, j):
                if PRECIOS[i][k] + TARIFAS[k][j] < MIN:
                    MIN = min(MIN, PRECIOS[i][k] + TARIFAS[k][j])
                RUTA[i][j] = k           #Anota que para ir de i a j hay que pasar por k
            PRECIOS[i][j] = MIN

    return PRECIOS,RUTA
#####

PRECIOS,RUTA = Precios(TARIFAS)
#print(PRECIOS[0][6])

print("PRECIOS")
for i in range(len(TARIFAS)):
    print(PRECIOS[i])

print("\nRUTA")
for i in range(len(TARIFAS)):
    print(RUTA[i])

#Determinar la ruta con Recursividad
def calcular_ruta(RUTA, desde, hasta):
    if desde == hasta:
        #print("Ir a :" + str(desde))
        return ""
    else:
        return str(calcular_ruta( RUTA, desde, RUTA[desde][hasta])) + \
               ',' + \
               str(RUTA[desde][hasta]) + \
               )

print("\nLa ruta es:")
calcular_ruta(RUTA, 0,6)

```

PRECIOS

```
[0, 5, 4, 3, 8, 8, 11]
[9999, 0, 999, 2, 3, 8, 7]
[9999, 9999, 0, 1, 6, 4, 7]
[9999, 9999, 9999, 0, 5, 6, 9]
[9999, 9999, 9999, 9999, 0, 999, 4]
[9999, 9999, 9999, 9999, 9999, 0, 3]
[9999, 9999, 9999, 9999, 9999, 9999]
```

RUTA

```
[0, 0, 0, 0, 1, 2, 5]
[', 1, 1, 1, 1, 3, 4]
[', ', 2, 2, 3, 2, 5]
[', ', ', 3, 3, 3, 3]
[', ', ', ', 4, 4, 4]
[', ', ', ', ', 5, 5]
[', ', ', ', ', ', ]
```

La ruta es:

Out[ ]: ',0,2,5'

## Ejercicio 5 (Opcional): Encontrar los dos puntos más cercanos (Aplicando distintas técnicas y para distintas dimensiones)

### *Descripción del problema:*

Problema: Encontrar los dos puntos más cercanos

Dado un conjunto de puntos se trata de encontrar los dos puntos más cercanos

Guía para aprendizaje:

- Suponer en 1D, o sea, una lista de números: [3403, 4537, 9089, 9746, 7259, ....]
- Primer intento: Fuerza bruta
- Calcular la complejidad. ¿Se puede mejorar?
- Segundo intento. Aplicar Divide y Vencerás
- Calcular la complejidad. ¿Se puede mejorar?
- Extender el algoritmo a 2D: [(1122, 6175), (135, 4076), (7296, 2741)....]
- Extender el algoritmo a 3D

In [ ]: `import random`

### Solución para 1D (Fuerza bruta)

```
In [ ]: def par_mas_cercano_1d_fuerza_bruta(puntos):
    min_distancia = 999999
    for i in range(len(puntos)):
        for j in range(i+1, len(puntos)):
            distancia = abs(puntos[i] - puntos[j])
            if distancia < min_distancia:
                min_distancia = distancia
                par_mas_cercano = (puntos[i], puntos[j])
```

```

    return par_mas_cercano, min_distancia

puntos = [random.randint(0, 100) for _ in range(10000)]
par, distancia = par_mas_cercano_1d_fuerza_bruta(puntos)

print(f"El par de puntos más cercano es {par} con una distancia de {distancia}")

```

El par de puntos más cercano es (77, 77) con una distancia de 0

La complejidad de este algoritmo es de  $O(n^2)$  ya que el primer bucle recorre toda la lista, es decir  $n$  veces, y el segundo bucle recorre la lista  $n - 1$  veces.

Las operaciones internas del bucle son comparaciones y asignaciones, por lo que la complejidad es de  $O(1)$ , ya que son operaciones de complejidad constante.

Por lo tanto, la complejidad total del algoritmo es de  $O(n^2)$ , dado que es  $n * (n - 1) * k = n^2 * k$ , siendo  $k$  constante.

## Solución para 1D (Divide y Vencerás)

```
In [ ]: def par_mas_cercano_1d_divide_y_vencerás(puntos):
    if len(puntos) <= 1:
        return None, 999999
    if len(puntos) == 2:
        return (puntos[0], puntos[1]), abs(puntos[0] - puntos[1])
    else:
        mitad = len(puntos) // 2
        izquierda = puntos[:mitad]
        derecha = puntos[mitad:]
        par_izquierda, distancia_izquierda = par_mas_cercano_1d_divide_y_vencerás(izquierda)
        par_derecha, distancia_derecha = par_mas_cercano_1d_divide_y_vencerás(derecha)
        distancia = min(distancia_izquierda, distancia_derecha)
        if distancia == distancia_izquierda:
            par = par_izquierda
        else:
            par = par_derecha
        for punto_izquierda in izquierda:
            for punto_derecha in derecha:
                if abs(punto_izquierda - punto_derecha) < distancia:
                    distancia = abs(punto_izquierda - punto_derecha)
                    par = (punto_izquierda, punto_derecha)
    return par, distancia

puntos = [random.randint(0, 100) for _ in range(10000)]
par, distancia = par_mas_cercano_1d_divide_y_vencerás(puntos)
print(f"El par de puntos más cercano es {par} con una distancia de {distancia}")

```

El par de puntos más cercano es (55, 55) con una distancia de 0

La complejidad de este algoritmo es de  $O(n \log n)$ , ya que se divide el problema en dos partes iguales en cada iteración, esto permite que la complejidad sea de  $O(\log n)$ , esto se repite  $n$  veces, por lo que la complejidad total es de  $O(n \log n)$ , ya que se reduce hasta la solución mínima que es la de dos elementos.

La operación de comparación de los dos puntos más cercanos tiene una complejidad de  $O(1)$ , ya que es una operación de complejidad constante.

Por lo tanto, la complejidad total del algoritmo es de  $O(n \log n)$ .

## ¿Se puede mejorar?

El problema es más eficiente aplicando la técnica de Divide y Vencerás, ya que la complejidad es menor que la de Fuerza Bruta.

Aún así, la complejidad de este algoritmo es de  $O(n \log n)$ , por lo que no es la solución más óptima para este problema.

Para llegar a una solución más óptima, se podría aplicar un algoritmo de programación dinámica, que podría permitiría reducir la complejidad

## Solución para 2D (Fuerza bruta)

```
In [ ]: # Distancia euclídea entre dos puntos de 2 dimensiones
def distancia_euclidea(p1, p2):
    return ((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)**0.5

def par_mas_cercano_2d_fuerza_bruta(puntos):
    min_distancia = 999999
    for i in range(len(puntos)):
        for j in range(i+1, len(puntos)):
            d = distancia_euclidea(puntos[i], puntos[j])
            if d < min_distancia:
                min_distancia = d
                par_mas_cercano = (puntos[i], puntos[j])
    return par_mas_cercano, min_distancia

puntos = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(10)]
par, distancia = par_mas_cercano_2d_fuerza_bruta(puntos)

print(f"El par de puntos más cercano es {par} con una distancia de {distancia}")
```

El par de puntos más cercano es ((70, 17), (74, 3)) con una distancia de 14.560219778561036

## Solución para 3D (Divide y Vencerás)

```
In [ ]: # Distancia euclídea entre dos puntos de 3 dimensiones
def distancia_euclidea(p1, p2):
    return ((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2 + (p1[2] - p2[2])**2)**0.5

def par_mas_cercano_3d_divide_y_vencerás(puntos):
    if len(puntos) <= 1:
        return None, 999999
    if len(puntos) == 2:
        return (puntos[0], puntos[1]), distancia_euclidea(puntos[0], puntos[1])
    else:
        puntos = sorted(puntos, key=lambda x: x[0])
        mitad = len(puntos) // 2
        izquierda = puntos[:mitad]
        derecha = puntos[mitad:]
        par_izquierda, distancia_izquierda = par_mas_cercano_3d_divide_y_vencerás(izquierda)
        par_derecha, distancia_derecha = par_mas_cercano_3d_divide_y_vencerás(derecha)
        distancia = min(distancia_izquierda, distancia_derecha)
        return (par_izquierda, par_derecha), distancia
```

```
if distancia == distancia_izquierda:
    par = par_izquierda
else:
    par = par_derecha
franja = [p for p in puntos if abs(p[0] - puntos[mitad][0]) < distancia]
for i in range(len(franja)):
    for j in range(i+1, len(franja)):
        if abs(franja[i][1] - franja[j][1]) < distancia:
            d = distancia_euclidea(franja[i], franja[j])
            if d < distancia:
                distancia = d
                par = (franja[i], franja[j])
return par, distancia

puntos = [(random.randint(0, 100), random.randint(0, 100), random.randint(0, 100))]
par, distancia = par_mas_cercano_3d_divide_y_venceras(puntos)
print(f"El par de puntos más cercano es {par} con una distancia de {distancia}")
```

El par de puntos más cercano es ((25, 95, 4), (25, 91, 7)) con una distancia de 5.0