



▼ Algoritmos: Actividad Guiada 1

David Bouzas i Morales

Martes, 20 de enero de 2026

Enlace a GitHub

https://github.com/19DavidBM/Algoritmos-de-Optimizacion/tree/main/Actividades_Guiadas/Actividad_1

▼ 1. Divide y Vencerás: Torres de Hanoi

El problema de las **Torres de Hanoi** consiste en trasladar una serie de discos de distinto tamaño apilados en un palo origen a un palo destino, utilizando un palo auxiliar, bajo la condición de que solo se puede mover un disco a la vez y ningún disco puede colocarse sobre otro más pequeño.

Este problema se puede resolver mediante un algoritmo de **Divide y Vencerás**, que consiste en descomponer el problema original en subproblemas más pequeños: primero se trasladan los discos superiores al palo auxiliar, luego se mueve el disco más grande al palo destino y, finalmente, se trasladan los discos del auxiliar al destino siguiendo la misma estrategia. De este modo, el algoritmo aplica recursión de manera sistemática, reduciendo problemas complejos a soluciones más sencillas y estructuradas.

A continuación, se muestra la implementación del método **Torres_Hanoi** usando el algoritmo de Divide y Vencerás. La idea principal es descomponer el problema de mover N discos en subproblemas más pequeños: si solo hay un disco, se mueve directamente del palo de origen al palo de destino; si hay más de uno, primero se trasladan los N-1 discos superiores al palo auxiliar, luego se mueve el disco más grande al destino, y finalmente se trasladan los N-1 discos del auxiliar al destino. Este proceso se aplica de manera recursiva hasta completar todos los movimientos, garantizando que se cumplan las reglas del juego:

```
# Método Torres de Hanoi usando Divide y Vencerás
def Torres_Hanoi(N, desde, hasta):
    # Caso base: si solo hay 1 disco, se mueve directamente del palo 'desde' al
    if N ==1 :
        print("Lleva la ficha ", desde, " hasta ", hasta)

    else:
        # Paso 1: mover N-1 discos del palo 'desde' al palo auxiliar
        Torres_Hanoi(N-1, desde, 6-desde-hasta)

        # Paso 2: mover el disco más grande (el N-ésimo) del palo 'desde' al pa
        print("Lleva la ficha ", desde, " hasta ", hasta)
```

```
# Paso 3: mover los N-1 discos del palo auxiliar al palo 'hasta'
Torres_Hanoi(N-1, 6-desde-hasta, hasta)
```

```
Torres_Hanoi(3, 1 , 3)
# -> Torres_Hanoi(2, 1 , 2)
# -> -> Torres_Hanoi(1, 1 , 3)
# -> -> -> N == 1 solo hará el print(Lleva la ficha Roja desde 1 hasta 3).
# -> -> Torres_Hanoi(1, 1 , 2)
# -> -> -> N == 1 solo hará el print(Lleva la ficha Amarilla desde 1 hasta 2).
# -> -> Torres_Hanoi(1, 3 , 2)
# -> -> -> N == 1 solo hará el print(Lleva la ficha Roja desde 3 hasta 2).
# -> Torres_Hanoi(2, 2 , 3)
# -> -> Torres_Hanoi(1, 2 , 1)
# -> -> -> N == 1 solo hará el print(Lleva la ficha Roja desde 3 hasta 2).
# -> -> Torres_Hanoi(1, 1 , 3)
```

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

Otro ejemplo en el que se puede aplicar el algoritmo de Divide y Vencerás es en el cálculo de la **sucesión de Fibonacci**. En este método, el problema de calcular el término N se divide en dos subproblemas más pequeños, correspondientes a los términos N-1 y N-2. El algoritmo se implementa de forma recursiva, estableciendo un caso base para los valores menores que 2 y un caso recursivo que combina los resultados de las dos llamadas anteriores:

```
# Método fibonacci usando Divide y Vencerás
def Fibonacci(N:int):
    # Caso base
    if N <= 2:
        return 1

    # Caso recursivo
    else:
        return Fibonacci(N-1) + Fibonacci(N-2)

Fibonacci(10)
```

55

Mejora Fibonacci: Exponenciación de Matrices con Divide y Vencerás

Aunque esta solución es totalmente válida, también es extremadamente ineficiente porque recalcula los mismos subproblemas múltiples veces. Esto genera un árbol de llamadas binario de profundidad n dando una complejidad temporal de $O(2^n)$. Pero este problema también puede resolverse tratando la sucesión de Fibonacci como una transformación lineal. En lugar de sumar números uno a uno, podemos expresar la relación entre términos consecutivos mediante una matriz:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Sea b la base (en nuestro caso, la matriz de Fibonacci) y n el exponente al que queremos elevar dicha base, el algoritmo de exponenciación binaria, también conocido como pow-and-square, se basa en la siguiente propiedad recursiva:

$$b^n = \begin{cases} (b^{n/2})^2 & \text{si } n \text{ es par} \\ b \cdot (b^{(n-1)/2})^2 & \text{si } n \text{ es impar} \end{cases}$$

Esta observación nos permite dividir el problema en mitades en cada paso. Matemáticamente, podemos definir la función recursiva para el cálculo de la potencia como:

$$f(b, n) = \begin{cases} 1 & \text{si } n = 0 \\ f(b, n/2)^2 & \text{si } n \text{ es par} \\ b \cdot f(b, (n-1)/2)^2 & \text{si } n \text{ es impar} \end{cases}$$

```
import numpy as np

# Método que implementa la Exponenciación Binaria usando el algoritmo de "cuadrado y multiplicar"
def matrix_power(matrix, power):
    # Matriz identidad del mismo tamaño
    result = np.identity(matrix.shape[0], dtype=object)
    base = matrix.copy()

    # Bucle en que cada iteración el exponente se divide entre 2 O(log)
    exp = power
    while exp > 0:
        # Si exponente impar, resultado se multiplica por la matriz actual
        if exp % 2 == 1:
            result = result @ base

        # División exponente a la mitad
        exp //= 2

        # Matriz base se eleva al cuadrado
        if exp > 0:
            base = base @ base

    return result

# Método fibonacci implementando la Exponenciación Binaria
def fibonacci_matrix(n):
    # Caso base
    if n == 0:
        return 0

    # Matriz base de Fibonacci
    matrix = np.array([[1, 1],
                      [1, 0]], dtype=object)

    # Exponenciación binaria implementada
    result = matrix_power(matrix, n-1)
    return result[0][0]
```

```
fibonacci_matrix(10)
```

55

La complejidad de este algoritmo viene determinada por el número de multiplicaciones de matrices que realiza la función `matrix_power()`. Al implementar la técnica de exponenciación binaria, el algoritmo no multiplica la matriz de forma secuencial, sino que aplica una estrategia de "divide y vencerás" donde el exponente n se reduce a la mitad en cada paso.

Matemáticamente, el número de veces que se puede dividir un valor n entre dos hasta llegar a la unidad es proporcional a su logaritmo en base 2 ($\log_2 n$). Como en cada una de estas etapas solo se realiza una multiplicación de matrices de tamaño fijo (2×2), el número total de operaciones crece de forma logarítmica respecto al valor de n .

▼ 2. Técnica Voraz: Devolución de cambio

El problema de la **devolución de cambio** consiste en entregar a un cliente una cantidad específica utilizando la menor cantidad posible de monedas disponibles. Cada moneda tiene un valor fijo, y el objetivo es seleccionar monedas de manera que la suma sea exacta y el número total de monedas sea mínimo.

Este problema se puede resolver mediante la **técnica voraz**, que consiste en tomar siempre la moneda de mayor valor posible que no exceda la cantidad restante a devolver, y repetir el proceso hasta completar el total. De este modo, la estrategia selecciona soluciones locales óptimas en cada paso con la esperanza de alcanzar la solución global de forma sencilla y eficiente.

A continuación, se muestra la implementación del método **cambio_monedas** usando la técnica voraz. La idea principal es seleccionar en cada paso la moneda más grande posible sin superar la cantidad restante por devolver, actualizando progresivamente el valor acumulado hasta alcanzar la cantidad total deseada. De este modo, el algoritmo toma decisiones localmente óptimas en cada iteración, reduciendo el problema de forma sencilla y eficiente y construyendo la solución final sin necesidad de explorar todas las combinaciones posibles:

```
# Método devolución de cambio usando técnica voraz
def cambio_monedas(N, SM):
    # Inicializar la solución con ceros (cada posición representa el número de
    SOLUCION = [0]*len(SM)

    # Almacenar la cantidad total devuelta hasta el momento
    ValorAcumulado = 0

    # Recorre el sistema de monedas ordenado de mayor a menor
    for i,valor in enumerate(SM):
        # Calcula el número máximo de monedas del valor actual que se pueden usar
        monedas = (N-ValorAcumulado)//valor

        # Guarda el número de monedas usadas de este valor
        SOLUCION[i] = monedas
```

```
# Actualiza el valor total devuelto
ValorAcumulado = ValorAcumulado + monedas*valor

# Si alcanza la cantidad exacta, se devuelve la solución
if ValorAcumulado == N:
    return SOLUCION

cambio_monedas(15, [25,10,5,1])

[0, 1, 1, 0]
```

Mejora de Solución en la Devolución de cambio: Programación Dinámica

Aunque el algoritmo voraz ofrece una solución eficiente recorriendo una única vez el sistema de monedas, no garantiza obtener el número mínimo de monedas en todos los casos. Para resolver este problema, se puede aplicar la técnica de programación dinámica, que almacena los resultados de subproblemas y construye la solución óptima de forma incremental, con una complejidad $O(N \cdot k)$. Esta solución que he planteado, aunque la complejidad sea mayor que la del enfoque voraz, garantiza encontrar siempre la solución óptima:

```
def cambio_monedas_pd(N, SM):
    # dp[x] = mínimo número de monedas para devolver cantidad x
    dp = [float('inf')] * (N + 1)
    dp[0] = 0

    # Para reconstruir la solución
    ultima_moneda = [-1] * (N + 1)

    # Construcción de la tabla DP
    for i, valor in enumerate(SM):
        for cantidad in range(valor, N + 1):
            if dp[cantidad - valor] + 1 < dp[cantidad]:
                dp[cantidad] = dp[cantidad - valor] + 1
                ultima_moneda[cantidad] = i

    # Si no hay solución
    if dp[N] == float('inf'):
        return None

    # Reconstrucción de la solución
    solucion = [0] * len(SM)
    cantidad = N
    while cantidad > 0:
        i = ultima_moneda[cantidad]
        solucion[i] += 1
        cantidad -= SM[i]

    return solucion

cambio_monedas_pd(15, [25,10,5,1])

[0, 1, 1, 0]
```

✓ 3. Técnica de Vuelta Atrás: N-Reinas

El problema de las **N-Reinas** consiste en colocar N reinas en un tablero de ajedrez de tamaño N × N de manera que ninguna reina ataque a otra, es decir, que no comparten la misma fila, columna ni diagonal. El objetivo es encontrar todas las posibles configuraciones que cumplan esta condición.

Este problema se puede resolver mediante la **técnica de vuelta atrás**, que consiste en colocar una reina en una fila y luego intentar colocar la siguiente en la fila siguiente verificando que no haya conflictos. Si en algún momento no es posible colocar una reina, el algoritmo retrocede (backtracking) y prueba una posición diferente para la reina anterior. De este modo, el método explora todas las combinaciones posibles, descartando aquellas que no cumplen las restricciones y construyendo únicamente soluciones válidas.

A continuación, se muestra la implementación del método **reinas** usando la técnica de vuelta atrás. La idea principal es colocar una reina por columna y comprobar en cada etapa si la solución parcial es prometedora, es decir, si no existen conflictos con las reinas ya colocadas. Cuando se detecta una posición inválida, el algoritmo retrocede (vuelta atrás) y prueba una alternativa distinta; en caso contrario, continúa avanzando hasta completar el tablero. De este modo, se exploran todas las configuraciones posibles, descartando aquellas que no cumplen las restricciones y construyendo únicamente soluciones válidas:

```

def escribe(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="")

# Método que comprueba si la solución parcial es válida hasta la etapa actual
def es_prometedora(SOLUCION, etapa):
    # Recorre las reinas colocadas hasta la etapa actual
    for i in range(etapa + 1):
        # Comprueba que no haya 2 reinas en la misma fila
        if SOLUCION.count(SOLUCION[i]) > 1:
            return False

    # Comprueba conflictos en las diagonales
    for j in range(i + 1, etapa + 1):
        # 2 reinas están en la misma diagonal si la diferencia de filas es
        if abs(i - j) == abs(SOLUCION[i] - SOLUCION[j]):
            return False

    # Si no hay conflictos, la solución es prometedora
    return True

# Método del problema de las reinas usando la técnica de vuelta atrás usando un
def reinas(N, solucion=[], etapa=0):

```

```

# Cada posición representa una columna y el valor indica la fila de la reina
if len(solucion) == 0:
    solucion = [0 for i in range(N)]

# Intenta colocar una reina en cada fila posible de la columna actual (etapa)
for i in range(1, N + 1):
    solucion[etapa] = i

    # Comprueba si la solución parcial es prometedora
    if es_prometedora(solucion, etapa):

        # Si se han colocado todas las reinas, se ha encontrado una solución
        if etapa == N - 1:
            print(solucion)
            escribe(solucion)
            print()
        else:
            # Continúa colocando la siguiente reina (siguiente columna)
            reinas(N, solucion, etapa + 1)

    # Vuelta atrás: se elimina la reina colocada en la etapa actual
    solucion[etapa] = 0

reinas(4)
# Respuesta 1: [2, 4, 1, 3] --> [(Columna 1, Fila 2), (Columna 2, Fila 4), (Columna 3, Fila 1), (Columna 4, Fila 3)]
# Respuesta 2: [3, 1, 4, 2] --> [(Columna 1, Fila 3), (Columna 2, Fila 1), (Columna 3, Fila 4), (Columna 4, Fila 2)]
```

[2, 4, 1, 3]

-	-	X	-
X	-	-	-
-	-	-	X
-	X	-	-

[3, 1, 4, 2]

-	X	-	-
-	-	-	X
X	-	-	-
-	-	X	-

▼ 4. Programación Dinámica: Viaje por el río

El problema del **Viaje por el río** consiste en encontrar la ruta óptima para cruzar un río, representado como una serie de posiciones o pasos, de manera que se minimice el costo total o se maximice la ganancia asociada a cada movimiento. Cada posición puede tener un valor distinto y existen restricciones sobre los pasos que se pueden dar para avanzar de un punto a otro.

Este problema se puede resolver mediante **programación dinámica**, que consiste en descomponer el problema en subproblemas más pequeños y almacenar sus soluciones para evitar cálculos repetidos. De esta manera, el algoritmo construye la solución óptima paso a paso, combinando resultados previos de manera eficiente y garantizando que se obtenga la ruta con el costo mínimo o la ganancia máxima sin necesidad de explorar todas las combinaciones posibles.

A continuación, se muestra la implementación del método **calcular_ruta** usando la programación dinámica. La idea principal es reconstruir el camino óptimo entre 2 nodos a partir de la información previamente almacenada en la matriz RUTA, la cual ha sido calculada mediante programación dinámica. El método utiliza recursividad para descomponer el recorrido en tramos más pequeños, siguiendo los nodos intermedios que forman parte del camino mínimo, hasta llegar al destino final. De este modo, se obtiene la secuencia completa de nodos que componen la ruta óptima sin necesidad de recalcular los costes:

```
# Matriz de tarifas entre nodos (TARIFAS[i][j] representa el costo de ir de i
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]
]

# Método de programación dinámica para calcular los precios mínimos
def Precios(TARIFAS):
    # Total de Nodos
    N = len(TARIFAS[0])

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

    # Inicialización de la tabla de rutas
    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
                    PRECIOS[i][j] = MIN

    return PRECIOS,RUTA

PRECIOS,RUTA = Precios(TARIFAS)

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

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

```
# Método para 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)

# [0, 0, 0, 0, 1, 2, 5] --> Nodos 1, 2, 3 tienen conexión directa a 0 (se les
# ['', 1, 1, 1, 1, 3, 4]
# ['', '', 2, 2, 3, 2, 5]
# ['', '', '', 3, 3, 3, 3]
# ['', '', '', '', 4, 4, 4]
# ['', '', '', '', '', 5, 5]
```

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, 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:

```
'0,2,5'
```