

✓ Algoritmo de Euclides para calcular el máximo común divisor (MCD)

```
def euclides(a, b):
    """
    Calcula el máximo común divisor (MCD) de dos números enteros a y b
    usando el Algoritmo de Euclides.
    """
    while b != 0:
        a, b = b, a % b
    return a

# Ejemplo de uso
num1 = int(input("Introduce el primer número: "))
num2 = int(input("Introduce el segundo número: "))

mcd = euclides(num1, num2)
print(f"El MCD de {num1} y {num2} es: {mcd}")
```

```
euclides(24, 12)
```

```
➞ Introduce el primer número: 2352342
Introduce el segundo número: 23545634
El MCD de 2352342 y 23545634 es: 2
12
```

✓ Decorador python para medir el tiempo de ejecución

```
import time

# Decorador para medir el tiempo de ejecución
def medir_tiempo(func):
    def wrapper(*args, **kwargs):
        inicio = time.perf_counter()
        resultado = func(*args, **kwargs)
        fin = time.perf_counter()
        print(f"Tiempo de ejecución de '{func.__name__}': {fin - inicio:.6f} segundos")
        return resultado
    return wrapper
```

✓ Método de Herón para aproximar la raíz cuadrada

```
@medir_tiempo
def raiz_cuadrada_heron(n, tolerancia=1e-10, max_iteraciones=1000):
    """
    Calcula la raíz cuadrada de un número n utilizando el Método de Herón.
    Parámetros:
    - n: Número del cual calcular la raíz cuadrada (debe ser >= 0).
    - tolerancia: Precisión deseada para la solución.
    - max_iteraciones: Número máximo de iteraciones permitidas.

    Retorna:
    - Una aproximación de la raíz cuadrada de n.
    """
    if n < 0:
        raise ValueError("No se puede calcular la raíz cuadrada de un número negativo.")
```

```

# Inicialización de la estimación (puede ser n o un valor aproximado inicial)
x = n if n != 0 else 0.0
iteraciones = 0

while iteraciones < max_iteraciones:
    # Nueva estimación según el método de Herón
    nuevo_x = 0.5 * (x + n / x)


    # Si la diferencia entre iteraciones es menor que la tolerancia, detenerse
    if abs(nuevo_x - x) < tolerancia:
        return nuevo_x

    x = nuevo_x
    iteraciones += 1

# Si no se alcanzó la tolerancia en el número máximo de iteraciones
raise RuntimeError(f"No se alcanzó la convergencia después de {max_iteraciones} iteraciones.")

# Ejemplo de uso
numero = float(input("Introduce el número para calcular su raíz cuadrada: "))
resultado = raiz_cuadrada_heron(numero)
print(f"La raíz cuadrada de {numero} es aproximadamente {resultado}")

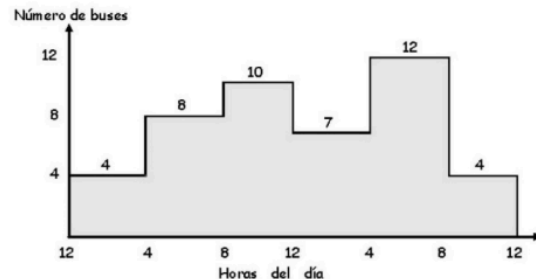
```

 Introduce el número para calcular su raíz cuadrada: 76
 Tiempo de ejecución de 'raiz_cuadrada_heron': 0.000036 segundos
 La raíz cuadrada de 76.0 es aproximadamente 8.717797887081346

✎ Algoritmos de Optimización. Componentes

Desarrollo e implementación de algoritmos. Ejemplo

Una pequeña ciudad estudia introducir un sistema de transporte urbano de autobuses. Nos encargan estudiar el **número mínimo de autobuses** para cubrir la demanda. Se ha realizado un estudio para estimar el número mínimo de autobuses por franja horaria. Lógicamente este número varia dependiendo de la hora del día. Se observa que es posible dividir la franja horaria de 24h en **tramos de 4 horas** en los queda determinado el número de autobuses que se necesitan. Debido a la normativa cada autobús debe circular **8 horas como máximo y seguidas** en cada jornada de 24h.



✎ Variables decisoras

```

from itertools import product

# Paso 1: Inicializamos los datos
# Demanda mínima de autobuses por tramo
demanda = [4, 8, 10, 7, 12, 4] # d[0], d[1], ..., d[5]
tramos = len(demanda) # Número de tramos (6 en este caso)

```

✚ Restricciones

```

#Posible Solucion
x = [4,5,6,7,8,9]

for t in range(tramos):
    # Calculamos el número actual de autobuses que están cubriendo el tramo t
    cobertura_actual = x[t] + x[t - 1] # Autobuses en t y t-1 (cíclico)

    # Si la cobertura actual es menor que la demanda, añadimos autobuses en t
    if cobertura_actual < demanda[t]:
        # Añadimos los autobuses necesarios en el tramo t
        x[t] += demanda[t] - cobertura_actual

```

✚ Función Objetivo

$$f(x) = \sum_{i=1}^6 x_i$$

```

#Función objetivo
f_objetivo = sum(x)
print(f_objetivo)

```

→ 39

Resolucion por fuerza bruta (evaluando todas las posibles soluciones)

- ✚ Para este caso por el numero de tramos horarios la ejecucion del codigo se tornara muy lenta por el numero de combinaciones que debe procesar.

```

from itertools import product
import numpy as np

def min_buses_bruteforce(demands, max_hours=8):
    num_slots = len(demands)
    max_slots = max_hours // 4
    max_buses = sum(demands)
    optimal_buses = max_buses
    optimal_assignment = None

    for num_buses in range(1, max_buses + 1):
        print(f"Probando configuraciones con {num_buses} autobuses...")
        assignments = product([0, 1], repeat=num_buses * num_slots)

        for assignment in assignments:
            assignment_matrix = np.array(assignment).reshape((num_buses, num_slots))

            # Evaluación temprana (pruning)
            if not np.all(assignment_matrix.sum(axis=0) >= demands):
                continue

            if not validate_assignment(assignment_matrix, demands, max_slots):
                continue

```

```

        # Contar cuántos autobuses están activos
        current_buses_used = np.sum(np.any(assignment_matrix, axis=1))
        if current_buses_used < optimal_buses:
            optimal_buses = current_buses_used
            optimal_assignment = assignment_matrix

    return optimal_buses, optimal_assignment

def validate_assignment(assignment_matrix, demands, max_slots):
    # Validar satisfacción de la demanda
    if not np.all(assignment_matrix.sum(axis=0) >= demands):
        return False

    # Validar franjas consecutivas
    for bus_schedule in assignment_matrix:
        if np.any(np.convolve(bus_schedule, np.ones(max_slots, dtype=int), mode='valid') > max_slots):
            return False

    return True

# Ejemplo de entrada

demands = [8, 10, 7, 12, 6, 9]
optimal_buses, optimal_assignment = min_buses_bruteforce(demands)

# Resultados
print(f"\nNúmero óptimo de autobuses: {optimal_buses}")
if optimal_assignment is not None:
    print("Matriz de asignación (autobuses x franjas horarias):")
    print(optimal_assignment)
else:
    print("No se ha encontrado ninguna asignación válida.")

```

✓ Complejidad:

- Si el problema crece en complejidad según la cantidad de tramos horarios
- ¿Que orden de complejidad tiene el problema(*)?

El algoritmo anterior tiene un orden de complejidad exponencial respecto a la cantidad de tramos horarios, esto debido a como se generan las combinaciones de asignaciones. Por lo que en este caso tenemos las variables relevantes las cuales serian:

n: Numero de tramos horarios

m: Numero de autobuses considerado

Este algoritmo evalua todas las posibilidades binarias de **m** autobuses en **n** tramos horarios. Es por esto que por cada autobus, puede estar asignado (1) o no (0) en cada tramo, por lo que esto generaria $2^{m \cdot n}$ combinaciones.

Teniendo en cuenta esto se puede decir que se obtendra una complejidad final del algoritmo que se puede expresar como:

$$O(2^{m \cdot n} * m * n)$$

Donde:

- $2^{m \cdot n}$ es el crecimiento exponencial de la busqueda
- $m * n$ es el tiempo de validacion por cada combinación