

Actividad Guiada 1 de Algoritmos de Optimización

Nombre: Aday Padilla Amaya

https://colab.research.google.com/drive/1x33Ki4YWoql1tXfXUrUzXL2AK5f_8Coe#scrollTo=GG2D6vLdYwwyhttps://github.com/mi_usuario/03MAIR--Algoritmos-de-Optimizacion

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

[] 1 celda oculta

✓ 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: 2
Tiempo de ejecución de 'raiz_cuadrada_heron': 0.000006 segundos
La raíz cuadrada de 2.0 es aproximadamente 1.414213562373095
```

Algoritmos de Optimización. Componentes

VC1 - Introducción a los algoritmos

Asignatura: Algoritmos de Optimización

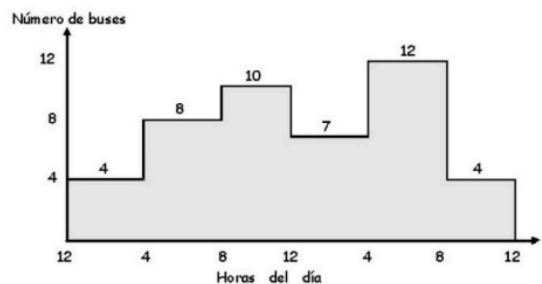
Desarrollo e implementación de algoritmos. Ejemplo

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.



viu Universidad Internacional de Valencia

32



Variables decisoras

```
from google.colab import drive
drive.mount('/content/drive')
```

```
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 = [4,4,4,4,4,8]

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)
```

 **Mostrar salida oculta**

```
import itertools
#Posible Solucion

x = [1,2,3,4,5,6]

# generar combinaciones
combinaciones = []
for r in range(1, 31):
    combinaciones.extend(itertools.combinations_with_replacement(x, r))

x = combinaciones
# método para fuerza bruta
def calcular_guagua(combinacion, demanda, tramos):
    x_ = [0] * tramos
    total_guagua = 0

    for i in range(len(combinacion)):
        x_[(i % tramos)] += combinacion[i]

    # ver si cubre la demanda
    for t in range(tramos):
        cobertura_actual = x_[(t % tramos)] + x_[(t - 1) % tramos]

        if cobertura_actual < demanda[t]:
            x_[(t % tramos)] += demanda[t] - cobertura_actual

        total_guagua += x_[(t % tramos)] # sumar guagua


    return total_guagua, x_

# fuerza bruta
mejor_combinacion = None
mejor_total_guagua = float('inf')
mejor_x = None

for combinacion in combinaciones:
    total_guagua, x_resultado = calcular_guagua(combinacion, demanda, tramos)

    # buscamos la guagua menor
    if total_guagua < mejor_total_guagua:
        mejor_total_guagua = total_guagua
        mejor_combinacion = combinacion
        mejor_x = x_resultado

print(f"Mejor reparto de guaguas: {mejor_x}")
print(f"Mínimo de guaguas: {mejor_total_guagua}")

 Mejor reparto de guaguas: [4, 4, 6, 1, 11, 0]
Mínimo de guaguas: 26
```

e tramos la combinacion puede crecer demasiado, por lo que la fuerza bruta será inviable, siendo una complejidad polinomial, $O(t)$ donde t