Actividad Guiada 1 de Algoritmos de Optimizacion

Nombre: Marcos Jose Diaz Gutierrez

https://colab.research.google.com/drive/1JF3LUCA5EZtCWcuJmsbsr-BT3_m0LwSO?usp=sharing

https://github.com/mi_usuario/03MAIR---Algoritmos-de-Optimizacion

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

In [ ]:
```python
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)
```

```
El MCD de 12 y 4 es: 4
```

Out[ ]: 12

Decorador python para medir el tiempo de ejecución

In [ ]:
```python
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 raiz cuadrada

In [ ]:
```python
@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}")
```

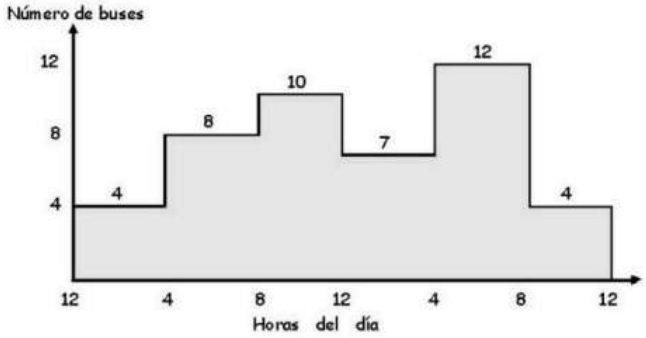
```
Tiempo de ejecución de 'raiz_cuadrada_heron': 0.000013 segundos
La raíz cuadrada de 45.0 es aproximadamente 6.708203932499369
```

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.

8 horas x día



viu Universidad Internacional de Valencia

32

Variables decisoras

In [ ]:
```
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

In [ ]:
```python
#Posible Solucion
x = [0,0,0,0,0,0]

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
    print(f'Tramo: {t}\nCobertura: {cobertura_actual}\nDemanda: {demanda[t]}\nx: {x}\n')
    if cobertura_actual < demanda[t]:
        # Añadimos los autobuses necesarios en el tramo t
        x[t] = x[t] + demanda[t] - cobertura_actual
    print(f'Tramo: {t}\nCobertura: {cobertura_actual}\nDemanda: {demanda[t]}\nx: {x}\n\n')

x
```

```
Tramo: 0
Cobertura: 0
Demanda: 4
x: [0, 0, 0, 0, 0, 0]

Tramo: 0
Cobertura: 0
Demanda: 4
x: [4, 0, 0, 0, 0, 0]


Tramo: 1
Cobertura: 4
Demanda: 8
x: [4, 0, 0, 0, 0, 0]

Tramo: 1
Cobertura: 4
Demanda: 8
x: [4, 4, 0, 0, 0, 0]


Tramo: 2
Cobertura: 4
Demanda: 10
x: [4, 4, 0, 0, 0, 0]

Tramo: 2
Cobertura: 4
Demanda: 10
x: [4, 4, 6, 0, 0, 0]


Tramo: 3
Cobertura: 6
Demanda: 7
x: [4, 4, 6, 0, 0, 0]

Tramo: 3
Cobertura: 6
Demanda: 7
x: [4, 4, 6, 1, 0, 0]
```

```
Tramo: 4
Cobertura: 1
Demanda: 12
x: [4, 4, 6, 1, 0, 0]

Tramo: 4
Cobertura: 1
Demanda: 12
x: [4, 4, 6, 1, 11, 0]


Tramo: 5
Cobertura: 11
Demanda: 4
x: [4, 4, 6, 1, 11, 0]

Tramo: 5
Cobertura: 11
Demanda: 4
x: [4, 4, 6, 1, 11, 0]
```

Out[ ]: [4, 4, 6, 1, 11, 0]

Función Objetivo

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

In [ ]:
```
#Función objetivo
f_objetivo = sum(x)
f_objetivo
```

Out[ ]: 26

Puntos Extra

In [ ]:
```python
lista_demandas = [4, 8, 10, 7, 12, 4]

def es_solucion_valida(solucion, lista_demandas):
    """
    Esta funcion toma una lista de 6 elementos tipo int y devuelve si es solucion al problema True o False
    """
    ans = True

    for t in range(len(solucion)):

        cobertura_actual = solucion[t] + solucion[t-1]

        if cobertura_actual < lista_demandas[t]:

            ans = False

            break

    return ans

# Test
print(es_solucion_valida(solucion=[0,0,0,0,0,0], lista_demandas=lista_demandas))
print(es_solucion_valida(solucion=lista_demandas, lista_demandas=lista_demandas))
```

```
False
True
```

In [ ]:
```python
def siguiente_lista(solucion, max_value=46):
    """
    Genera la siguiente lista en orden ascendente según las reglas de incremento.

    Parámetros:
    - solucion: lista actual de 6 elementos.
    - max_value: valor máximo permitido para cada elemento (por defecto 46).

    Retorna:
```

```
    - Una nueva lista con la siguiente combinación.
    """
    for i in range(len(solucion) - 1, -1, -1):  # Recorre desde el último elemento
        # print(f'i: {i}, solucion[i]: {solucion[i]}\n')
        solucion[i] += 1  # Incrementa el elemento actual
        if solucion[i] <= max_value:  # Si no excede el máximo, termina
            break
        solucion[i] = 0  # Si excede el máximo, lo resetea a 0 y sigue al siguiente
    return solucion

# Test
print(siguiente_lista([0, 0, 0, 0, 0, 0], max_value=46))
print(siguiente_lista([0, 0, 0, 0, 0, 46], max_value=46))
print(siguiente_lista([46, 46, 46, 46, 46, 45], max_value=46))
```

```
[0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0]
[46, 46, 46, 46, 46, 46]
```

In [ ]:
```
def hallar_total_buses(lista_demandas):

    solucion = [0, 0, 0, 0, 0, 0]

    solucion_final = (solucion, 46)

    i = 0

    while True:

        # Si encontramos una solucion mejor, sustituimos la actual
        if es_solucion_valida(solucion, lista_demandas) and sum(solucion) < solucion_final[1]:
            solucion_final = (solucion, sum(solucion))
            print(f'({i}/({46**6}) Nueva solucion encontrada:\n{solucion_final} - {solucion_final[1]}')

        # Actualizamos el valor de solucion
        solucion = siguiente_lista(solucion, max_value=46)

        # Chequeamos que no hayamos llegado al ultimo intento
        if solucion == [46] * 6:
            break

        i += 1
```

```
    return solucion_final

solucion_final = hallar_total_buses(lista_demandas=lista_demandas)
```

```
(39256472/(9474296896) Nueva solucion encontrada:
([0, 8, 2, 5, 7, 4], 26) - 26
```

Cálculo de la complejidad del algoritmo

1. **Validación de cada solución (`es_solucion_valida`)**:
   - Comprueba si una lista de 6 elementos cumple las restricciones del problema.
   - Complejidad: (O(1)) (constante, porque la lista tiene longitud fija).
2. **Generación de la siguiente combinación (`siguiente_lista`)**:
   - Incrementa los valores de la lista para generar la siguiente combinación.
   - Complejidad: (O(1)) (constante, porque la lista tiene longitud fija).
3. **Iteración principal (`while True`)**:
   - Explora todas las posibles combinaciones de una lista de 6 elementos donde cada uno puede tomar valores entre 0 y 46 ((47) valores).
   - Número total de combinaciones: (47^6).
   - Complejidad: (O(47^6)).

Complejidad total

La complejidad está dominada por el número de iteraciones, por lo que el algoritmo tiene una **complejidad exponencial**: [O(47^6).]

Complejidad si crece el número de tramos

El oren de complejidad del problema probando 47 valores por tramo y teniendo 6 tramos era de $O(47^6)$ por lo que si tuviéramos un número, n, de tramos y quisiéramos probar, p, posibles valores por cada tramo tendríamos una complejidad computacional de $O(p^n)$. O, equivalentemente, una complejidad exponencial en términos de la cantidad de tramos.