# actividadguiada1

July 11, 2025

# 1 Algoritmos - Actividad Guiada 1

**Nombre:** Elvis David Pachacama **URL:** https://github.com/ElvisDavis/maestria-algoritmos/blob/main/actividadguiada1.ipynb

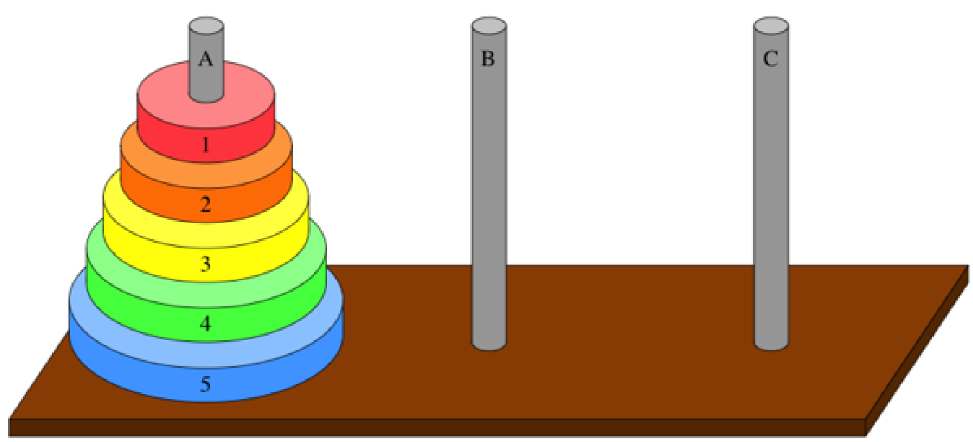
# 2 Torres de hanoi

## 2.1 Algoritos recursivos

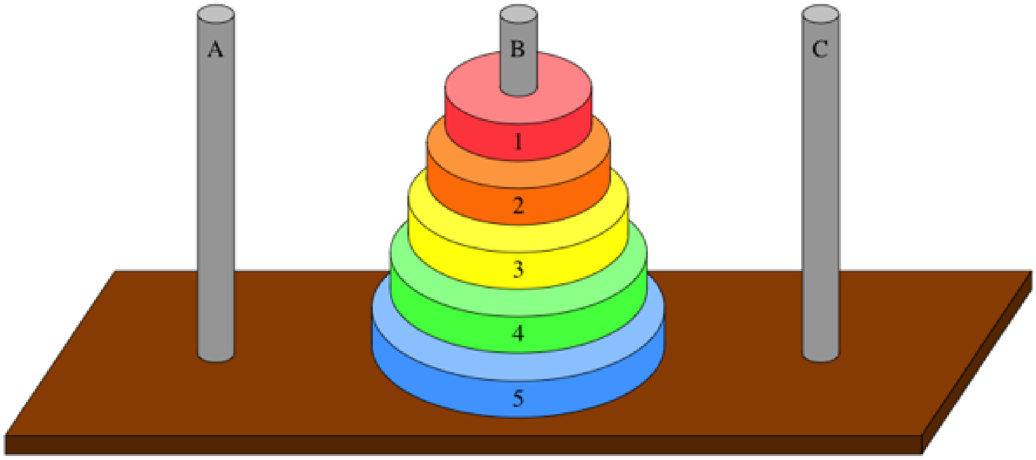
Un algoritmo recursivo es un método de programación en que la función se invoca a si isma para resolver una versión reducida del mismo problema. Este proceso continúa hasta que el problema se simplifica lo suficinete como para resolverse directamente, sin necesidad de invocaciones adicionales. La idea clave de la recursión es que el problema se divide en partes más pequeñas y manejables, lo que hace más facil resolverlo.

### 2.1.1 Ejercicio Torres de Hanoi

Se llama las **Torres de hanoi**. Te dan un conjunto de tres varillas y n discos, con cada disco de un tamaño diferente. LLamaremos a las varillas A, B, y numeramos los discos desde 1, siendo el disco más pequeño, hasta n, siendo el disco más grande. Al principio, todos los n discos están en la varilla A, en orden de tamaño decreciente de la parte inferior a la parte superior, de modo que el disco n está en la parte inferior y el disco 1 está en la parte superior. Aquí esta como se ven las torres de Hanoi para n=4 discos:



El objetivo del ejercicio es pasar todos los n discos de la varilla A a la varilla B.




**Debemos seguir las siguientes reglas:** * Podemos mover solamanete un disco a la vez * Ningún disco puede estar encima de un disco más pequeño. Por ejemplo, si el disco 3 está en una varilla, entonces todos los discos debajo del disco 3 deben tener números mayores que 3.

### 2.2 Pasos a seguir

1. Primero, vamos a ver cómo resolver el problema de manera recursiva. Vamos a empezar con un caso realmente sencillo: un disco, es decir **n=1**. El caso de **n=1** será nuestro caso base. Siempre puedes mover el disco1 de la varilla A a la varilla B, porqu sabes que cualquier disco debajo debe ser mayor. Y no hay nada especial acerca de las varillas A y B. Puedes mover el disco de la varilla B a la varilla C si lo deseas, o de la varilla C a la varilla A, o de cualquier varilla. Resolver el problema de las Torres de Hanoi con un deisco es trivial, y requiere mover el único disco solamente una vez.

[1]:
```python
def Torres_Hanoi(N, desde, hasta):
    if N==1:
        print("LLeva la ficha desde " , desde , "hasta " , hasta)
    else:
        #TorresHahoi(N-1, desde, 6-desde-hasta)
        Torres_Hanoi(N-1, desde, 6-desde-hasta)# 6-desde-hasta calculo la torre␣
 ↪pivot
        print("LLeva la ficha ", desde , "hasta", hasta)
        #Torres_hanoi(N-1,6-desde-hasta, hasta)
        Torres_Hanoi(N-1, 6-desde-hasta, hasta)
```

[2]:
```python
Torres_Hanoi(3, 1, 3)
```

```
LLeva la ficha desde  1 hasta  3
LLeva la ficha  1 hasta 2
LLeva la ficha desde  3 hasta  2
LLeva la ficha  1 hasta 3
LLeva la ficha desde  2 hasta  1
```

```
LLeva la ficha  2 hasta 3
LLeva la ficha desde  1 hasta  3
```

[3]:

```python
import time
#Representamos las torres como pilas
torres={
    1:[],
    2:[],
    3:[]
}
torres
```

[3]:

```
{1: [], 2: [], 3: []}
```

[4]:

```python
#creamo una función donde se imprime cada una de las torres recibe un parametro␣
 ↪torres
def imprimir_torres(torres):
    print("\nEstado actual de las torres:")
    for i in range(1,4):
        print(f"Torre {i}: {torres[i]}")
    print("-" *30)
    time.sleep(1)
```

[5]:

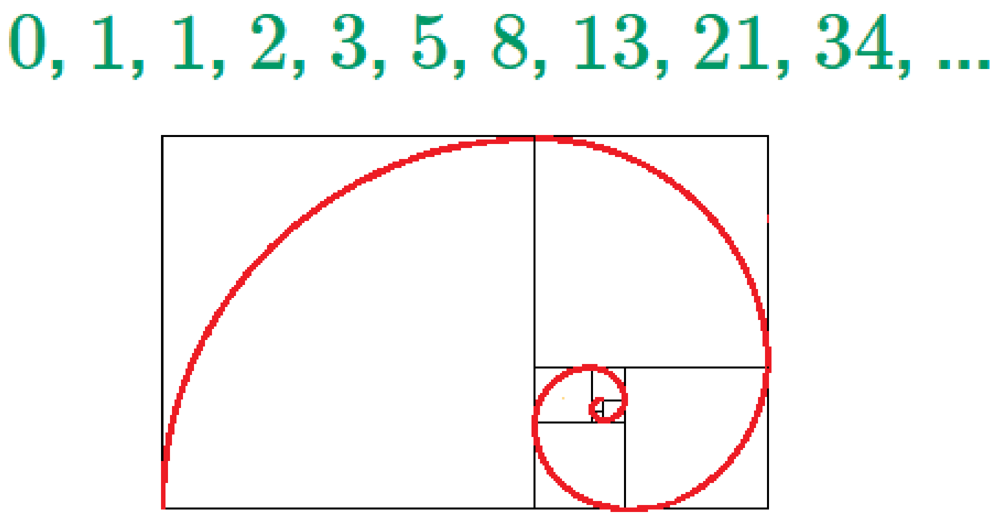
```python
def mover_ficha(torres, origen, destino):
    ficha = torres[origen].pop()
    torres[destino].append(ficha)
    print(f"Moviendo ficha {ficha} de torre {origen} a torre {destino}")
    imprimir_torres(torres)  # o imprimir_torres_visual(torres, num_fichas)
```

[6]:

```python
def torres_hanoi_visual(n, origen, destino, auxiliar, torres):
    if n==1:
        mover_ficha(torres, origen, destino)
    else:
        torres_hanoi_visual(n-1, origen, auxiliar,destino, torres)
        mover_ficha(torres, origen, destino)
        torres_hanoi_visual(n-1, auxiliar, destino, origen, torres)
```

#Inicializamos 3 discos en la torre 1 num_fichas=5 torres[1]= list(reversed(range(1, num_fichas+1))) torres[2]=[] torres[3]=[] print("Inicio de las Torres de Hanoi") imprimir_torres(torres) torres_hanoi_visual(num_fichas, 1, 3, 2, torres) print("fin del juego")

## 3 Sucesión de Fibonacci




La sucesión de Fibonacci es conocida desde hace miles de años, pero fue Fibonacci (Leonardo de Pisa) quien le dio a conocer al utilizarla para resolver un problema. El **primer** y **segundo** término de la sucesión son:

$$a_0 = 0$$

$$a_1 = 1$$

Los siguientes términos se obtienen sumando los dos términos que les preceden: El **tercer término** de la sucesión es

$$a_2 = a_0 + a_1 =$$

$$= 0 + 1 = 1$$

El **cuarto término** es:

$$a_3 = a_1 + a_2 =$$

$$= 1 + 1 = 2$$

El **quinto término** es

$$a_4 = a_2 + a_3 =$$

$$= 1 + 2 = 3$$

El **sexto término** es:

$$a_5 = a_3 + a_4 =$$

$$= 2 + 3 = 5$$

El $(n+1)$ **-ésimo término** es:

$$a_n = a_n - 2 + a_n - 1$$

## Término general La sucesión de Fibonacci es una sucesión definida por recurrencia. Esto significa que para calcular un término de sucesión se necesitan los términos que le preceden. Se proporcionan los dos primeros términos: $a_0 = 0$ y $a_1 = 1$. Los siguinetes se calculan con la siguinete fórmula:

$$a_n + 1 = a_n - 1 + a_n, n >= 1$$

**Nota:** el primer término que proporciona la fórmula es $a_2$ (porque $n$ tiene que ser mayor o igual que **1**). Por esta razón, se definen $a_0$ y $a_1$ con anterioridad.

[7]:
```python
#Calculo del término n-simo de la sucesión de Fibonacci
def Fibonacci(N:int):
    if N<2:
        return 1
    else:
        return Fibonacci(N-1)+Fibonacci(N-2)
```

[8]:
```python
Fibonacci(5)
```

[8]: 8

[9]:
```python
# Función para generar la serie de Fibonacci hasta el término n
def serie_fibonacci(N):
    serie = []
    for i in range (N + 1):
        serie.append(Fibonacci(i))
    return serie
```

[10]:
```python
# Implementamos una función para imprimir la serie de fibonacci en seriepiramide
def imprimir_serie_fibonacci(N):
    serie = serie_fibonacci(N)
    for i in range(1, len(serie) +1):
        print (" " * (len(serie)-i), *serie[:i])
```

[11]:
```python
#Ejecutamos el código
n=8
print(f"Serie de Fibonacci {n}: ")
imprimir_serie_fibonacci(n)
```

```
Serie de Fibonacci 8:
         1
        1 1
       1 1 2
      1 1 2 3
     1 1 2 3 5
    1 1 2 3 5 8
   1 1 2 3 5 8 13
  1 1 2 3 5 8 13 21
 1 1 2 3 5 8 13 21 34
```

# 4 Fórmula de Binet

**La fórmula de Binet** es una fórula explicita y cerrada que se utiliza para hallar rl $n$ término n-ésimo de la sucesión de Fibonacci. Recibe este nombre porque fue derivada por el matemático

Jacques Philippe Marie Binet, aunque ya era conocidapor Abraham de Moivre.

**Fórmula** Si $F_n$ es el $n^0$ número de Fibonacci, entonces:

$$F_n = \frac{1}{\sqrt{a}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

[12]:
```
#importamos la libreria math
import math
def formula_binet(n):
    #calculamos el número áureo
    phi = (1 + math.sqrt(5)) / 2
    psi = (1 - math.sqrt(5)) / 2

    resultado = (phi**n - psi**n) / math.sqrt(5)
    return round(resultado)
```

[13]:
```
#implementamos una función que imprime los primeros n niveles en forma de␣
 ↪piramide
def piramide(numero_niveles):
    #Calculamos lso términos de Fibonnaci necesario
    total_numeros = numero_niveles * (numero_niveles +1)//2
    fibos=[formula_binet(i) for i in range(total_numeros)]
    index = 0
    for fila in range(1, numero_niveles+1):
        #espacios
        print (" " * (numero_niveles-fila) * 3, end="")
        for _ in range(fila):
            print(f"{fibos[index]:<4}", end=" ")
            index+=1
        print()
```

[14]:
```
piramide(5)
```

```
            0
        1     1
     2     3     5
   8     13    21    34
55    89    144   233   377
```

**Nota**Como se observa en relación de la primer aimplemntación, esta segunda lo que hace la fórmula de Binet es devolver el n-ésimo npumero Fibonacci desde F(0)

## 5 Devolución de cambio por técnica voraz

Un algoritmo voraz(greddy) es un algoritmo que encuentra una solución globalmente óptima a un problema a basde de hacer elecciones localmente óptimas. Es decir: el algoritmo siempre hace lo que "parece" mejor en cada momento, sin tener nunca que reconsiderar sus decisiones, y acaba llegando directamente a la mejor solución posible.

[15]:
```
def cambio_moneda(N, SM):
    SOLUCION = [0] * len(SM)
    ValorAcumulado= 0

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

        if ValorAcumulado == N:
            return SOLUCION
```

[16]:
```
cambio_moneda(15,[10,5,1,25])
```

[16]:
```
[1, 1, 0, 0]
```

Para complementar el concepto de algoritmos voraces eh conseguido un ejercicio. Hay $M(M <= 10^5)$ farolas en la posisición $y_1, ..., y_M$ de una recta y $N(N <= 10^5)$ puntos $x_1, ..., x_N$. Cada farola tiene un radio de iluminación $r_i$, tal que la i-ésima farola ilumina puntos en el intervalo $[y_i - r_i, y_i + r_i]$. Se requiere encender el mínimo número de farolas tales que cada uno de los $N$ puntos $x_1, ...x_N$ esté iluminado por al menos una farola. Encuentra este mínimo número.

### 5.0.1 Solución

- Se convierte cada farola en un intervalo de cobertura: intervalo= $[y_i - r_i, y_i + r_i]$
- Ordenamos estos intervalos por el extremo izquierdo o por el final
- Ordenamos los puntos x.
- Para cada punto x:
  - Mientras haya farolas cuyo intervalo **empieza antes o justo en x**, guardamos la que más lejos llegue
  - Encendemos esa farola, y saltamos al siguiente punto que no esté cubierto poe ella.

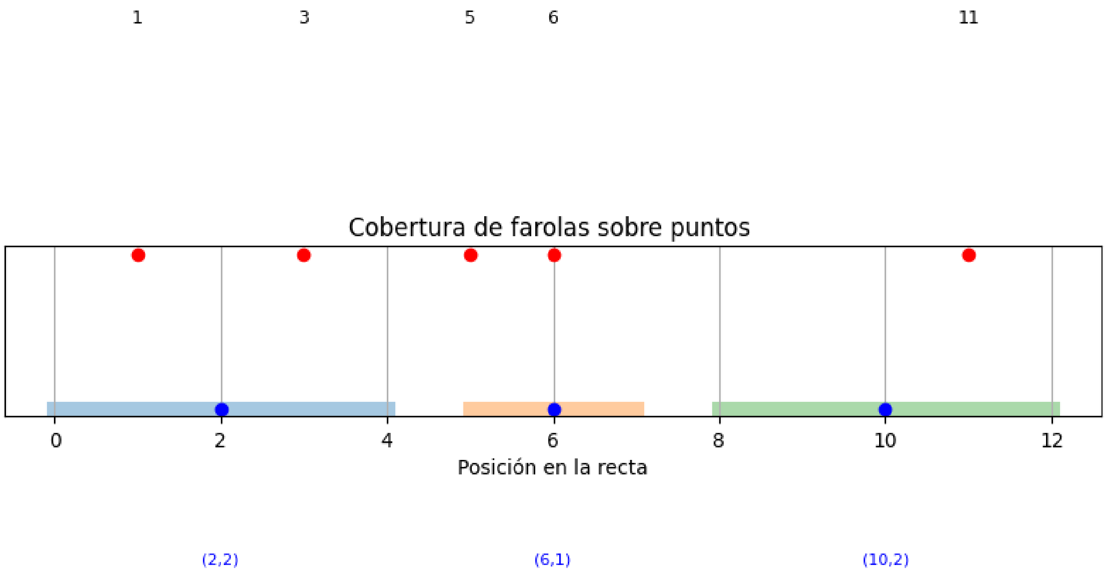
[17]:
```
#Importamos la biblioteca de matplotlib
import matplotlib.pyplot as plt
#Implementamos una función
def min_farolas(puntos, farolas):
    #Convertimos las farolas en intervalos [inicio y fin]
    intervalos = []
    for idx, (y, r) in enumerate(farolas):
        #añadimos los intervalos a las farolas
        intervalos.append((y-r,y+r,idx))
    # Ordenamos los intervalos por el inicio
    intervalos.sort()

    #Ordenamos los puntos que se deben cubrir
    puntos.sort()
```

```python
    #Declaramos e inicializamos el nuemro de farolas encendidas
    resultado=0
    encendidas=[]
    #Inicializamos el indice de lso intervalos
    i=0
    #Sacamos la longitud de los intervalos y lo guardamos en una variable n
    n=len(intervalos)
    #Inicializamos los indices para los puntos
    j=0
    #Creamos un cliclo repetitivo
    while j < len(puntos):
        x=puntos[j]
        max_cobertura = -1
        mejor_farola=-1

        #Buscamos la mejor farola que cubrea c
        while i < n and intervalos[i][0] <=x:
            if intervalos[i][1] > max_cobertura:
                max_cobertura= max(max_cobertura, intervalos[i][1])
                # Guardamos el indice
                mejor_farola=intervalos[i][2]
            i+=1
        if max_cobertura <x:
            #No hay ninguna farola que cubra el punto x
            return -1, [] # No se puede cubrir todos los puntos
        #Encedemos una farola que cubre hasta max-cobertura
        resultado += 1
        encendidas.append(farolas[mejor_farola])
        #saltamos todos lo puntos ya cubiertos
        while j< len(puntos) and puntos[j] <= max_cobertura:
            j+=1

    return resultado, encendidas
```

[18]:
```python
# 3 farolas  con sus posiciones y radios
farolas = [(2,2), (6,1), (10,2)]

# puntos que deben ser cubiertos
puntos = [1,3,5,6,11]

minimo, usadas = min_farolas(puntos, farolas)
print("Minimo de farolas", minimo)
print("Farolas utilizadas: ")
for pos, radio in usadas:
    print(f" Posición: {pos}, Radio: {radio}")
```

```
Minimo de farolas 3
Farolas utilizadas:
 Posición: 2, Radio: 2
 Posición: 6, Radio: 1
 Posición: 10, Radio: 2
```

[19]:
```
# Visualización gráfica
fig, ax = plt.subplots(figsize=(10,2))
#Dibujamos los puntos rojo
for p in puntos:
    # Punto a cubrir
    #Colocamos los puntos ligeramente arriba de la linea
    ax.plot(p, 0.02, 'ro', zorder=3)
    ax.text(p, 0.05, f"{p}", ha='center', fontsize=9, zorder=4)

#dibujar farolas como intervalos
for pos, radio in usadas:
    inicio = pos - radio
    fin = pos + radio
    #barra de cobertura
    ax.plot([inicio,fin],[0,0], lw=8, alpha=0.4, zorder=1)
    #centro de la farola
    ax.plot(pos, 0, 'bo', zorder=2)
    ax.text(pos, -0.02, f"({pos},{radio})", ha='center', fontsize=8,␣
 ↪color='blue', zorder=2)

#Configuramos el gráfico
ax.set_yticks([])
ax.set_xlabel("Posición en la recta")
ax.set_title("Cobertura de farolas sobre puntos ")
ax.grid(True)
plt.subplots_adjust(top=0.85, bottom=0.25)
plt.show()
```




### 5.1 ¿Por qué es una solución voraz?

Porque en cada se elige la mejor opción local posible (la farola que más cubre el siguiente punto no cubierto), sin irar al futuro. En este tipo de problemas, la elección local resulta ser globalmente óptima, siempre y cuando lso intervalos estén ordenados.

## 6 N-Reinas por técnica vuelta atrás

#### 6.0.1 ¿Qué es la Técnica de Vuelta Atrás?

La vuelta atrás (backtracking) es una estrategia de resolución de problemas basada en la búsqueda sistemática de soluciones en un espacio de decisiones. Se utiliza cuando el problema requiere construir una solución paso a paso, y en cada paso se deben validar ciertas condiciones para determinar si se puede continuar o si es necesario retroceder(backtrack) y probar otras alternativas. Es una técnica especialmente útil en problemas que involucran: * Combinatorias (permutaciones, combinaciones) * Configuraciones (como tableros, secuencias o asignaciones) * Satisfacción de restricciones (como en juegos, puzzles o tareas de asiganción)

### 6.1 Principios de Funcionamiento

EL algoritmo de vuelta atrás explora todas las posibles soluciones de manera estrucutrada utilizando recursión. En cada paso. * Se genera una decisión parcil. * Se verifica si es válida bajo las restricciones del problema. * Se la solución parcial es válida, se continúa con la siguiente decisión. * Si se detecta que no puede conducir a una solución completa válida, se deshace ( se retorcede) y se prueba otra opción * Este proceso se asemeja a un árbol de decisión donde el algoritmo recorre ramas y vueleve al nodo anterior cuando encuentra un camino invalido.

## 7 Caracteristicas del Algoritmo

- **Recursivo:** la solución se construyr paso a paso
- **Eficiente** frente a fuerza bruta, ya que descarta cambios no viables temprano
- **Explícitamente controlado:**

## 8 Descripción del Problema

El problema de las N-Reinas consiste en ubicar N reinas en tablero de agedrez d NxN, d emodo que ninguna reina ataque a otra. En términos prácticos, se deben cumplir las siguientes condiciones:

- Ninguna reina debe compartir la misma fila.
- Ninguna reina debe compartir la misma columna.
- Ninguna reina debe ubicarse en la mima dagonal qeu otra. Este es un problema clásico de satisfacción e restricciones, ideal para ser resuleto mediante la técnica de vuelta atrás (backtracking)

# Estructura Lógica del Algoritmo

El algoritmo se implementa en tres funciones principales:

[20]:
```
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="")
```

[21]:
```
def es_prometedora(SOLUCION, etapa):
    #Si la solución tiene dos valores iguales no es valida => Dos reinas en la␣
 ↪misma fila
    for i in range(etapa+1):
        if SOLUCION.count(SOLUCION[i])>1:
            return False

        #Verifica las diagonales
        for j in range (i+1, etapa +1):
            if abs(i-j) == abs(SOLUCION[i]-SOLUCION[j]) :
                return False
    return True
```

[22]:
```
def reinas(N, solucion=[], etapa=0):
    if len(solucion)==0:
        solucion =[0 for i in range (N)]
```

```python
    for i in range(1, N+1):
        solucion[etapa]=i

        if es_prometedora(solucion, etapa):
            if etapa == N-1:
                print(solucion)
                print()
            else:
                reinas(N, solucion, etapa+1)
        else:
            None
        solucion[etapa]=0
```

[23]:
```python
reinas(8)
```

```
[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]
```

# 9 Viaje por el rio. Programación dinámica

[10]:

```
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]]

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 pasa por 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")
for i in range(len(TARIFAS)):
```

```python
    print(PRECIOS[i])

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

#Determinamos 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, 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 :
Ir a :0
```

[10]: ',0,2,5'

[14]:
```python
import matplotlib.pyplot as plt
import networkx as nx
```

[15]:

```
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]]

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 pasa por 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


#Determinamos la ruta con Recursividad
def calcular_ruta_lista(RUTA, desde, hasta):
    if desde == hasta:
        return [desde]
    else:
        return calcular_ruta_lista(RUTA, desde, RUTA[desde][hasta]) + [hasta]
```

[16]:

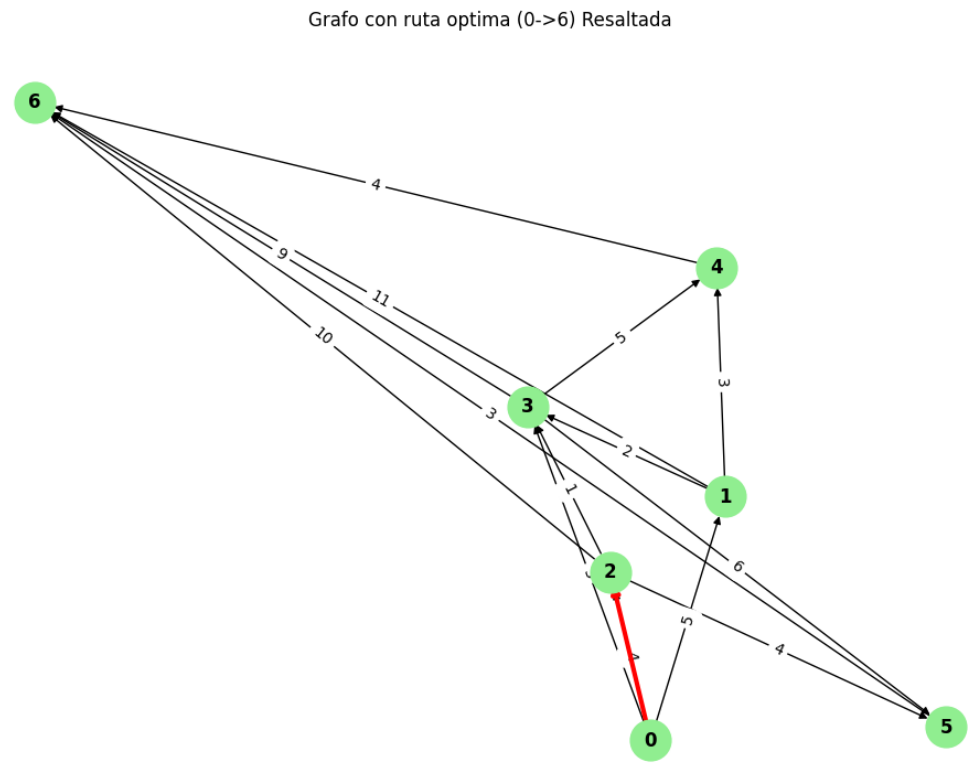
```
#Construcción del grafo
G = nx.DiGraph()
N = len(TARIFAS)
for i in range(N):
    for j in range (N):
        if TARIFAS[i][j] != 999 and i != j:
```

```
            G.add_edge(i, j, weight=TARIFAS[i][j])
```

[17]:
```
# Obtener ruta y graficar
PRECIOS, RUTA = Precios(TARIFAS)
ruta_optima = calcular_ruta_lista(RUTA, 0, 6)
```

[18]:
```
#Posiciones de los nodos
pos = nx.spring_layout(G, seed=42)
```

[23]:
```
# Dibujar todos los nodos y aristas
plt.figure(figsize=(10, 7))
nx.draw(G, pos, with_labels=True, node_size=700, node_color="lightgreen",␣
 ↪font_size=12, font_weight="bold", arrows=True)

#Dibujar etiquetas en los pesos
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

#Dibujar la ruta optima en rojo
ruta_edges = list(zip(ruta_optima[:1], ruta_optima[1:]))
nx.draw_networkx_edges(G, pos, edgelist=ruta_edges, edge_color="red", width=3)

plt.title("Grafo con ruta optima (0->6) Resaltada")
plt.axis('off')
#plt.tight_layout()
plt.show()
```




[22]:
```
print("\nLa ruta es :")
calcular_ruta_lista(RUTA, 0,6)
```

```
La ruta es :
```

[22]:
```
[0, 2, 5, 6]
```

[ ]: