Algoritmos - Actividad Guiada 1

Nombre: Alberto Paparelli
URL:
https://colab.research.google.com/drive/1GRAkJz2Ek21bX6AliGqj0wr_7MFtXJgP?usp=sharing

https://github.com/carpe-diem/VIU-03MIAR-algoritmos-de-optimizacion/blob/main/AG1-Actividad-Guiada-1/Algoritmos_AG1-Alberto_Paparelli.ipynb

Torres de Hanoi con Divide y vencerás

Mejoras

Se agrego una representacion visual del estado de las torres despues de cada movimiento. La funcion principal Torres_Hanoi(N, desde, hasta) mantiene la misma estructura recursiva original, reemplazando unicamente el print por una llamada a mover_y_dibujar, que realiza:

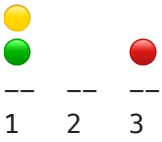
- Identificacion de fichas por color: Cada disco tiene un color asignado (disco 1 = roja, disco 2 = amarilla, disco 3 = verde), permitiendo seguir visualmente el recorrido de cada ficha individual a lo largo de la ejecucion.
- Dibujo del estado: Tras cada movimiento se muestra una representacion grafica de las 3 torres con los discos apilados, lo que facilita la comprension paso a paso del algoritmo.

La complejidad algoritmica no cambia, ya que la mejora es puramente de visualizacion.

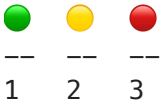
In [84]:
```
torres = {1: [], 2: [], 3: []}

def mover_y_dibujar(desde, hasta, N):
    COLORES = {1: "🔴", 2: "🟡", 3: "🟢"}
    NOMBRES = {1: "Roja", 2: "Amarilla", 3: "Verde"}
    disco = torres[desde].pop()
    print(f"Lleva la ficha {NOMBRES[disco]} desde {desde} hasta {hasta}")
    torres[hasta].append(disco)
    for fila in range(N, 0, -1):
        linea = ""
        for t in [1, 2, 3]:
            if fila <= len(torres[t]):
                linea += f"{COLORES[torres[t][fila-1]]}  "
            else:
                linea += "    "
        print(linea)
    print("--  " * 3)
    print("1   2   3")
    print()
```

```
def Torres_Hanoi(N, desde, hasta):
  if N == 1 :
    mover_y_dibujar(desde, hasta, 3)
  else:
    Torres_Hanoi(N-1, desde, 6-desde-hasta)
    mover_y_dibujar(desde, hasta, 3)
    Torres_Hanoi(N-1, 6-desde-hasta, hasta)

torres[1] = [3, 2, 1]
Torres_Hanoi(3, 1 , 3)
```

```
Lleva la ficha Roja desde 1 hasta 3
```



```
--  --  --
1   2   3

Lleva la ficha Amarilla desde 1 hasta 2
```

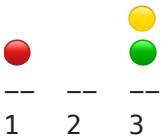


```
--  --  --
1   2   3

Lleva la ficha Roja desde 3 hasta 2
```

```
--  --  --
1   2   3

Lleva la ficha Verde desde 1 hasta 3
```

```
--  --  --
1   2   3

Lleva la ficha Roja desde 2 hasta 1
```

```
--  --  --
1   2   3

Lleva la ficha Amarilla desde 2 hasta 3
```



```
--  --  --
1   2   3

Lleva la ficha Roja desde 1 hasta 3
```

```
--  --  --
1   2   3
```

Suscesión de Fibonacci

Mejoras

- Se agregó una versión iterativa Fibonacci_PD() que utiliza una tabla para almacenar sub-problemas ya resueltos, evitando los recálculos redundantes de la versión recursiva.

- Se incluye una comparación de tiempos de ejecución con time para evidenciar la diferencia entre O(2^N) (recursiva) y O(N) (programación dinámica).

In [85]:
```python
import time

#Sucesión_de_Fibonacci
#https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci
#Calculo del termino n-simo de la suscesión de Fibonacci
def Fibonacci(N:int):
  if N < 2:
    return 1
  else:
    return Fibonacci(N-1)+Fibonacci(N-2)

def Fibonacci_PD(N: int):
    # Tabla para almacenar resultados
    tabla = [0] * (N + 1)
    tabla[0] = 1
    tabla[1] = 1

    for i in range(2, N + 1):
        tabla[i] = tabla[i-1] + tabla[i-2]

    return tabla[N]

# Comparar tiempos con N=30
N = 30

inicio = time.time()
resultado_rec = Fibonacci(N)
tiempo_rec = time.time() - inicio

inicio = time.time()
resultado_pd = Fibonacci_PD(N)
tiempo_pd = time.time() - inicio

print(f"Fibonacci({N}) = {resultado_rec}")
print(f"Recursiva:             {tiempo_rec:.6f} seg")
print(f"Programacion dinamica: {tiempo_pd:.6f} seg")
print(f"La PD es {tiempo_rec/tiempo_pd:.0f}x mas rapida")
```

```
Fibonacci(30) = 1346269
Recursiva:             0.085928 seg
Programacion dinamica: 0.000031 seg
La PD es 2772x mas rapida
```

Devolución de cambio por técnica voraz

In [86]:
```python
def cambio_monedas(N, SM):
  SOLUCION = [0]*len(SM)   #SOLUCION = [0,0,0,0,..]
  ValorAcumulado = 0

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

    if ValorAcumulado == N:
```

```
      return SOLUCION


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

Out[86]: [0, 1, 1, 0]

N-Reinas por técnica de vueta atrás

Mejoras

- Poda temprana: es_prometedora ahora solo compara la reina nueva (en etapa) contra las anteriores con range(etapa), en vez de recorrer toda la solución.
- Se elimina count(): Se reemplazó SOLUCION.count() > 1 por una comparación directa SOLUCION[i] == SOLUCION[etapa], pasando de O(N) a O(1) por verificación
- BUG: Fix mutable default: solucion=[] cambiado a solucion=None con if solucion is None. En Python, los argumentos por defecto se evalúan una sola vez cuando se define la función, no cada vez que se llama. Eso significa que todas las llamadas comparten la misma lista.

Ejemplo del bug:

```
def agregar(valor, lista=[]):
    lista.append(valor)
    return lista

print(agregar(1))  # [1]       - ok
print(agregar(2))  # [1, 2]    - debería ser [2]
print(agregar(3))  # [1, 2, 3] - debería ser [3]
La lista [] se crea una sola vez en memoria. Cada llamada
a agregar() modifica la misma lista.
```

In [ ]:
```
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="")


def es_prometedora(SOLUCION, etapa):
  for i in range(etapa):
    # Misma fila
    if SOLUCION[i] == SOLUCION[etapa]:
      return False
    # Misma diagonal
    if abs(i - etapa) == abs(SOLUCION[i] - SOLUCION[etapa]):
      return False
  return True
```

```
def reinas(N, solucion=None, etapa=0):
  if solucion is None:
      solucion = [0 for i in range(N)]

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

    if es_prometedora(solucion, etapa):
      if etapa == N-1:
        print(solucion)
        escribe(solucion)
        print()
      else:
        reinas(N, solucion, etapa+1)
    else:
      None

    solucion[etapa] = 0

reinas(4)
```

```
[2, 4, 1, 3]

 -  -  X  -
 X  -  -  -
 -  -  -  X
 -  X  -  -
[3, 1, 4, 2]

 -  X  -  -
 -  -  -  X
 X  -  -  -
 -  -  X  -
```

Viaje por el rio. Programación dinámica

Mejoras

- Reemplazo de magic numbers: Se reemplazaron los valores arbitrarios 999 y 9999 por float('inf') (infinito de Python).
- Inicializacion clara: Se cambio `[9999]*N for i in [9999]*N` por `[float('inf')]*N for _ in range(N)`. El codigo original usaba una lista `[9999]*N` como iterador del list comprehension, lo cual funcionaba (iteraba N veces porque la lista tenia N elementos) pero `range(N)` expresa mejor que se quieren N filas.
- Fix de calcular_ruta: La funcion original mezclaba print() con return de strings, produciendo una salida incorrecta: imprimia "Ir a :0" y retornaba ',0,2,5'. Lo cambié para que retorne una lista de paradas [0, 2, 5, 6] usando recursion, y luego se imprime con " -> ".join() para mostrar la ruta completa: 0 -> 2 -> 5 -> 6.

In [88]:
```
INF = float('inf')

TARIFAS = [
[0,5,4,3,INF,INF,INF],
```

```
[INF,0,INF,2,3,INF,11],
[INF,INF, 0,1,INF,4,10],
[INF,INF,INF, 0,5,6,9],
[INF,INF, INF,INF,0,INF,4],
[INF,INF, INF,INF,INF,0,3],
[INF,INF,INF,INF,INF,INF,0]
]



#####################################################################
def Precios(TARIFAS):
#####################################################################
  #Total de Nodos
  N = len(TARIFAS[0])

  #Inicialización de la tabla de precios
  PRECIOS = [ [float('inf')]*N for _ in range(N)]
  RUTA = [ [""]*N for _ in range(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 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 p
        PRECIOS[i][j] = MIN

  return PRECIOS,RUTA
#####################################################################


PRECIOS,RUTA = Precios(TARIFAS)

print(f"Precio minimo de 0 a 6: {PRECIOS[0][6]}")

print("\nTabla de PRECIOS (precio minimo para ir de i a j):")
for i in range(len(TARIFAS)):
  print(PRECIOS[i])

print("\nTabla de RUTA (ultimo nodo intermedio para ir de i a j):")
for i in range(len(TARIFAS)):
  print(RUTA[i])

#Determinar la ruta con Recursividad
def calcular_ruta(RUTA, desde, hasta):
  if desde == hasta:
    return [desde]
  else:
    intermedio = RUTA[desde][hasta]
    return calcular_ruta(RUTA, desde, intermedio) + [hasta]

print("\nRuta optima de 0 a 6:")
ruta = calcular_ruta(RUTA, 0, 6)
print(" -> ".join(str(x) for x in ruta))
```

```
Precio minimo de 0 a 6: 11

Tabla de PRECIOS (precio minimo para ir de i a j):
[0, 5, 4, 3, 8, 8, 11]
[inf, 0, inf, 2, 3, 8, 7]
[inf, inf, 0, 1, 6, 4, 7]
[inf, inf, inf, 0, 5, 6, 9]
[inf, inf, inf, inf, 0, inf, 4]
[inf, inf, inf, inf, inf, 0, 3]
[inf, inf, inf, inf, inf, inf, inf]

Tabla de RUTA (ultimo nodo intermedio para ir de i a j):
[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]
['', '', '', '', '', '', '']

Ruta optima de 0 a 6:
0 -> 2 -> 5 -> 6
```