

Algoritmos y Estructuras de Datos

LABORATORIO N° 02

Recursividad y Backtracking

CODIGO DEL CURSO:

<i>Alumno(s)</i>		<i>Nota</i>
<i>Huayroccacya Taco Nilton</i>		
<i>Grupo</i>	<i>C42 c</i>	
<i>Ciclo</i>	<i>III</i>	
<i>Fecha de entrega</i>	<i>26/03/2024</i>	

I.- OBJETIVOS:

- Definir las reglas básicas a seguir para la construcción y la correcta interpretación de los Diagramas de Flujo, resaltando las situaciones en que pueden, o deben, ser utilizados.
- Elaborar y Diseñar algoritmos con arreglos de una sola dimensión(unidimensional) denominada vectores

II.- SEGURIDAD:**Advertencia:**

En este laboratorio está prohibida la manipulación del hardware, conexiones eléctricas o de red; así como la ingestión de alimentos o bebidas.

III.- FUNDAMENTO TEÓRICO:

- Revisar el texto guía que está en el campus Virtual.

IV.- NORMAS EMPLEADAS:

- No aplica

V.- RECURSOS:

- En este laboratorio cada alumno trabajará con un equipo con Windows 10.

VI.- METODOLOGÍA PARA EL DESARROLLO DE LA TAREA:

- El desarrollo del laboratorio es individual.

VII.- PROCEDIMIENTO:**EJERCICIO DE APLICACIÓN****1. Recursividad : Factorial**

```
# Factorial function
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)

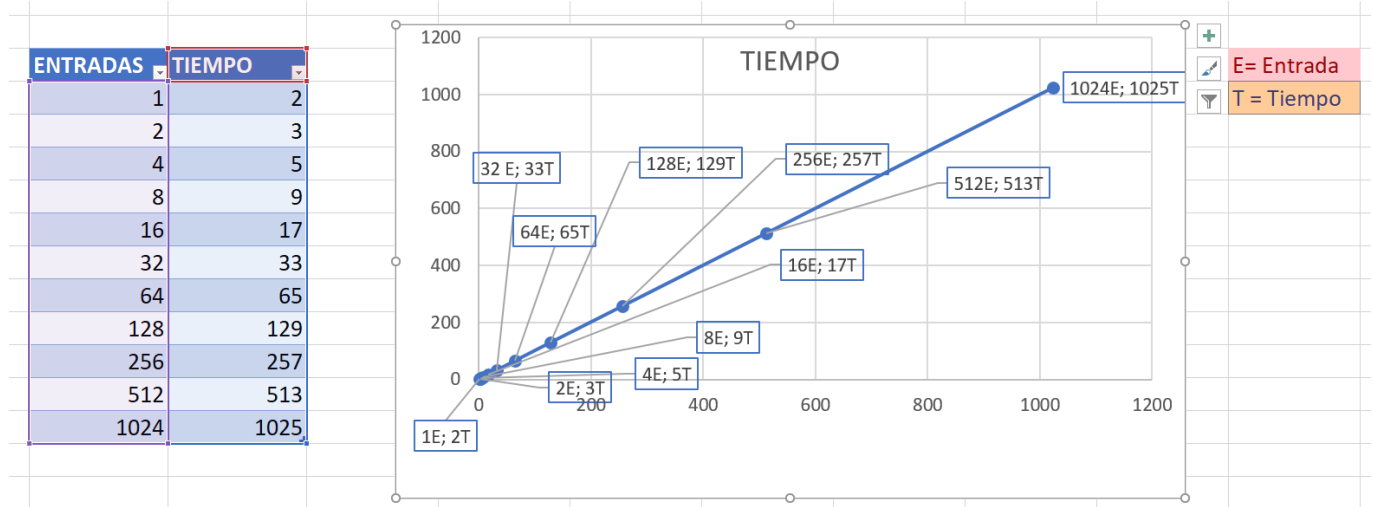
# Using
if __name__ == "__main__":
    print(factorial(3))
```

Mejoramos el código con comentarios

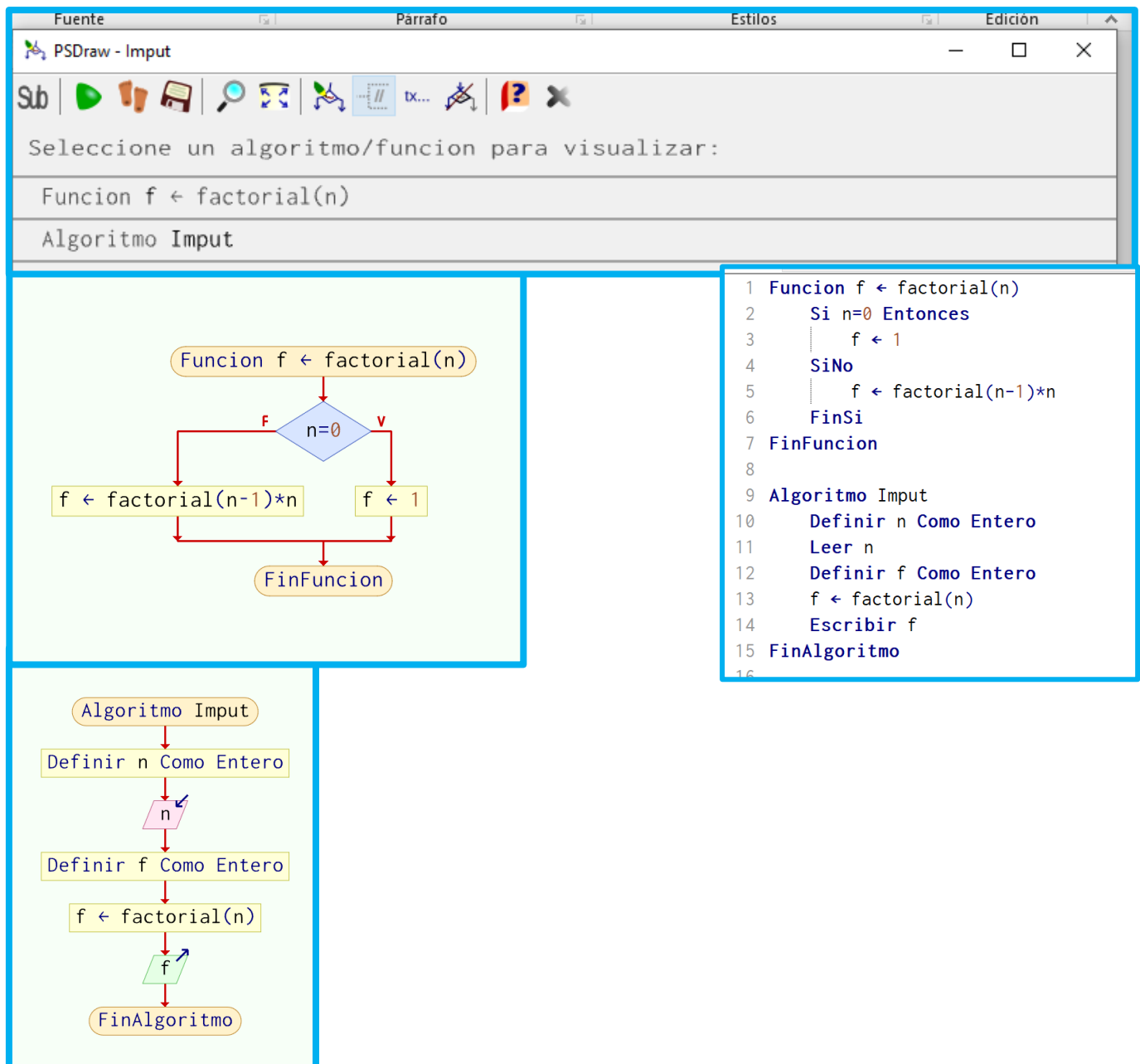
```
# Función factorial
def factorial(n):
    # Caso base: el factorial de 0 es 1
    if n == 0:
        return 1
    else:
        # Caso recursivo: n!
        return n * factorial(n - 1)

if __name__ == "__main__":
    # Calcula y muestra el factorial de 3
    print(factorial(3))
```

a.- Graficar el tiempo que demora en ejecutarse una función factorial para los valores de: 1,2,4,8,16,32,64,128,256,512,1024



b.- Implementar el diagrama de flujo



c.- Indicar sus apreciaciones

1: La tendencia que muestra la gráfica sugiere que el tiempo de ejecución aumenta a medida que el valor de entrada (y, por lo tanto, el tamaño del problema) aumenta. Esto es consistente con lo que se esperaría para una función factorial recursiva, ya que su complejidad es exponencial.

2: Se podría añadir una comprobación para asegurar que el argumento n es un entero no negativo, ya que el factorial no está definido para números negativos.

3: Cuando se trata de hallar el factorial o problemas de con la misma naturaleza es bueno usar recursividad y no un bucle for.

d.- Evaluar si es posible aumentar el tamaño del stack.

Si es posible aumentar el tamaño de la pila, tendríamos que usar el módulo sys para establecer un nuevo límite, con la función "setrecursionlimit". Sin embargo esto no se recomienda en muchos casos ya que puede sobrecargar tu memoria del sistema y eso no sería bueno.

```
11 import sys
12 sys.setrecursionlimit(10000) #Ajusta el nuevo límite a 100000
```

2. Recursividad : Torre de Hanoi

```
# Hanoi Function
def towersOfHanoi(numberOfDisks, src=1, dest=3, tmp=2):
    if numberOfDisks: towersOfHanoi(numberOfDisks-1, src = src, dest = tmp, tmp = dest)
        #print("Move disk %d from peg %d to peg %d"
        #      % (numberOfDisks, src, dest))
        towersOfHanoi(numberOfDisks-1, src = tmp, dest = dest, tmp = src)

if __name__ == "__main__":
    # Execute
    towersOfHanoi(numberOfDisks=6)
```

a.-Explicar como funciona el algoritmo

La función towersOfHanoi se llama a sí misma para mover los discos entre las estacas, reduciendo el problema en cada paso al mover todos los discos excepto el más grande a la estaca auxiliar, trasladar el disco más grande al destino final y luego apilar los discos menores encima del más grande en la ubicación final, este proceso se repite hasta que todos los discos se han movido correctamente a la estaca de destino.

**b.-Grafique el tiempo que demora en resolverse una torre de Hanoi de :
1,2,3,4,5,6,7,8,9,10,11,12,13,14**



c.- Indiques sus apreciaciones

1. El código es un claro ejemplo de cómo se puede utilizar la recursividad para resolver un problema complejo de manera elegante y concisa. La recursividad descompone el problema en instancias más pequeñas del mismo problema.
2. Estructura del código: El uso de `if __name__ == "__main__":` es una buena práctica en Python para asegurar que el código se ejecute solo cuando el archivo se ejecute como script principal y no cuando se importe como un módulo en otro archivo.
3. `towersOfHanoi` utiliza parámetros con valores predeterminados, lo que permite al usuario llamar a la función con solo el número de discos, asumiendo que las estacas de origen, destino y temporal son 1, 3 y 2.

3. Backtracking

```
# bitStrings Function
def bitStrings(n):
    if n == 0:
        return []
    if n == 1:
        return ["0", "1"]
    return [digit + bitstring
            for digit in bitStrings(1)
            for bitstring in bitStrings(n-1)]

if __name__ == "__main__":
    # Excuting
    print(bitStrings(3))
```

a.- Como funciona el algoritmo

Primero le damos como parámetro 3 a la función **bitStrings()** esto no dará un lista bits de longitud 3, como no cumple las dos primeras condiciones por lo que entra por default al ultimo return donde encontramos un tipo de lista llamo "list comprehension" o comprensión de lista, estando ahí encontramos 2 ciclos for anidados, donde en el primer **for digit** toma como valor 0 luego 1, en el **segundo for** comienza la llamada recursión, ya que **bitStrings(3-1)**, como 3-1 es 2 entonces llama de nuevo a toda la función donde el **primero for digit** toma el valor de nuevo 0 pero para **bitStrings(2-1) = 1**, una vez tengamos **bitStrings(1)** ya sabemos que nos devolverá ["0","1"] de acuerdo a las combinaciones de arriba ahora **bitstring** toma el valor de 0 y recordemos que **digit** también tomaba 0, entonces en la lista se concatenarán "00" luego **bitstring** toma 1 entonces "01" luego termina el **segundo for** pasa al primero for donde **digit** toma esta vez el valor de 1, formando así otra combinación con el segundo **for**. Luego que esto termine recordemos que todo esto es solo para **bitStrings(2)** dentro de **bitStrings(3)** ahora si recién regresamos a **bitStrings(3)**, donde de nuevo no cumple con las condicione del pricipio y se pasa al ultimo **return**, en el primer **for digit** toma el valor de "0", pasa al segundo for donde **bitstrings(3-2)** es 2 y nosotros ya tenemos **bitStrings(2)** que [00,01,10,11] y asi concatenamos para para digit 0 y par digit 1 donde nos darán un lista de bits de todas las combinaciones de longitud 3.

b.- Calcule el valor de biStrings para : 3,4,5,6

```
main.py > ...
1 def CadenaBits(num):
2     if num == 0:
3         return []
4     if num == 1:
5         return ["0", "1"]
6     return [
7         digit + bit for digit in CadenaBits(1)
8         for bit in CadenaBits(num - 1)
9     ]
10
11 print(CadenaBits(3))
12
```

```
--> poetry lock --no-update
Resolving dependencies...

Run
Ask AI 613ms on 14:22:22, 03/22 ✓

['000', '001', '010', '011', '100', '101', '110', '111']
```

```
main.py > f CadenaBits > ...
1 def CadenaBits(num):
2     if num == 0:
3         return []
4     if num == 1:
5         return ["0", "1"]
6     return [digit + bit for digit in
7         CadenaBits(1) for bit in CadenaBits(num-1)]
8     print(CadenaBits(4))
9
```

```
--> poetry lock --no-update
Resolving dependencies...

Run
Ask AI 43ms on 14:21:09, 03/22 ✓

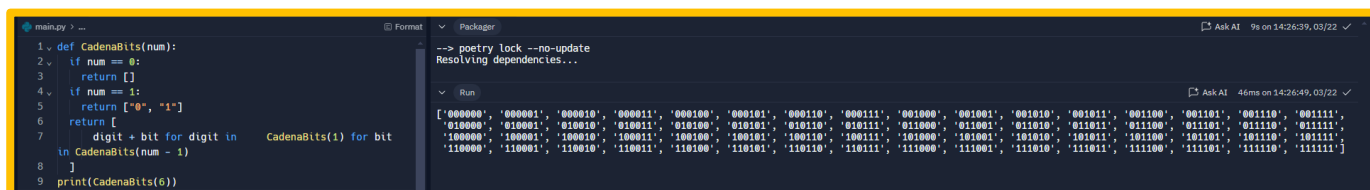
['0000', '0001', '0010', '0011', '0100', '0101', '0110',
 '1', '1111']
```

```
main.py > ...
1 def CadenaBits(num):
2     if num == 0:
3         return []
4     if num == 1:
5         return ["0", "1"]
6     return [
7         digit + bit for digit in CadenaBits(1) for bit in
8         CadenaBits(num - 1)
9     ]
10 print(CadenaBits(5))
11
```

```
--> poetry lock --no-update
Resolving dependencies...

Run
Ask AI 103ms on 14:23:47, 03/22 ✓

['01000', '01001', '01010', '01011', '01100', '01101', '01110', '01111',
 '10000', '10001', '10010', '10011', '10100', '10101', '10110', '10111',
 '11000', '11001', '11010', '11011', '11100', '11101', '11110', '11111']
```



```

1 def CadenaBits(num):
2     if num == 0:
3         return []
4     if num == 1:
5         return ["0", "1"]
6     return [
7         digit + bit for digit in CadenaBits(1) for bit
8         in CadenaBits(num - 1)
9     ]
10 print(CadenaBits(6))
  
```

```

--> poetry lock --no-update
Resolving dependencies...

Run

['000000', '000001', '000010', '000011', '000100', '000101', '000110', '000111', '001000', '001001', '001010', '001011', '001100', '001101', '001110', '001111', '010000', '010001', '010010', '010011', '010100', '010101', '010110', '010111', '011000', '011001', '011010', '011011', '011100', '011101', '011110', '011111', '100000', '100001', '100010', '100011', '100100', '100101', '100110', '100111', '101000', '101001', '101010', '101011', '101100', '101101', '101110', '101111', '110000', '110001', '110010', '110011', '110100', '110101', '110110', '110111', '111000', '111001', '111010', '111011', '111100', '111101', '111110', '111111']
  
```

c.- Indique sus apreciaciones

1. En cuanto si avise sido $\text{num} = 4$ o $\text{num} = n$, sucedería lo mismo en cada recursión el parámetro o **n** en esto caso va disminuyendo en 1 ya que **bitStrings(n-1)** entonces entraremos en recursión **hasta que n-1 = 1** y es ahí cuando cumplimos uno de los casos base y rompemos la recursión, finalmente tenemos que regresar hacia n ósea tenemos que volver a evaluar nuestra función en uno en uno: cuando llegamos **bitStrings(2)** hera parte de bitStrings(3) y bitStrings(3) hera parte de bitStrings(4) y así regresamos hasta llegar a **bitStrings(n)**. Esta funciona como una caja dentro de la caja hay otra caja mas pequeña y dentro de esa caja hay otra caja pequeña y para cerrar todas las cajas tenemos que comenzar por la mas pequeña hacia la mas grande.
2. Aunque el código es elegante y conciso, no es el más eficiente para generar cadenas de bits debido a la naturaleza recursiva y las múltiples llamadas a la función para $n == 1$. Podría optimizarse para evitar cálculos redundantes
3. La función hace llamadas redundantes a bitStrings(1) en cada iteración. Dado que el resultado de bitStrings(1) es siempre ["0", "1"], se puede almacenar este resultado en una variable antes del bucle para evitar llamadas innecesarias
4. Lo que yo creo que para que se entienda al principio mejor se tendría que comentar un poco más.

5. Version mejorada para evitar confuciones al principio:

Al cambiar `for digito in bitStrings(1)` por `for digito in ['0', '1']`, evitas llamar innecesariamente a la función bitStrings(1) en cada iteración del bucle, ya que el resultado de bitStrings(1) es siempre ['0', '1']

Sin esta asignación, la función bitStrings(n - 1) se llamaría repetidamente para cada dígito en ['0', '1']. Al almacenar el resultado en `cadenas_mas_cortas`, solo necesitas calcular las cadenas de bits de longitud n - 1 una vez, y luego puedes reutilizarlas.

```

def bitStrings(n):
    if n == 0:
        return [""]
    if n == 1:
        return ['0', '1']
    # Almacenar el resultado de bitStrings() hace que no llamemos muchas veces bitStrings(1)
    cadenas_mas_cortas = bitStrings(n - 1)
    # Cambiamos aquí for digito in bitStrings() por [0,1] para que no vuelva a llamar la funcion
    # inecesariamente.

    return [digito + cadena for digito in ['0', '1'] for cadena in cadenas_mas_cortas]
if __name__ == "__main__":
    print(bitStrings(3))
  
```

CONCLUSIONES:

1. La recursión, al principio, puede ser un concepto difícil de entender al menos el lo personal, ya que su funcionamiento no es tan directo como el de un bucle for. Sin embargo, después de analizar el problema y entender que cada llamada recursiva es un paso hacia el caso base, se hace evidente que es simplemente otra forma de iteración la recursión termina cuando se cumple una condición específica, similar a cómo un bucle for termina cuando se alcanza el final de un rango o una colección.
2. Como bien sabemos ahora que la recursión es especialmente útil en problemas que tienen una estructura naturalmente jerárquica o que se pueden dividir en problemas más pequeños de la misma naturaleza. Esto se ve claramente en el problema de las Torres de Hanoi, donde cada movimiento de discos se reduce a mover un disco menos que el anterior, demostrando que la recursión puede aportar a la solución de problemas complejos.
3. Cuando se trate de recursión es importante de pensar en términos de casos base y casos recursivos al diseñar una función recursiva. El caso base actúa como un ancla que evita que la recursión se extienda al infinito, mientras que el caso recursivo debe acercarse al caso base en cada llamada. Esta comprensión me ha permitido apreciar cómo se estructuran las funciones recursivas.
4. En el ejercicio de 3 de combinaciones de bits, estuve analizando el código línea por línea que también me tomo tiempo en asimilarlo pero sin darme cuenta al analizar este código estuve también haciendo backtracking, esto me permitió comprender que el backtracking es una estrategia eficaz para explorar todas las posibles soluciones de un problema, en este caso, la generación de todas las cadenas de bits de una longitud como parámetro.
5. Por último utilizamos algunos conceptos nuevos como las listas comprensión junto con 2 bucles for anidados, sin darnos cuenta también estamos aprendiendo más de estructuras de datos, esto nos ayudara ha comprender más acerca de las estructuras de datos en futuro.