	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I. Marco Antonio Martínez Quintana.

Asignatura: Estructura de Datos y Algoritmos I.

Grupo: 17

No de Práctica(s): 12

Integrante(s): Acosta Rodríguez Eder Alberto.

No. de Equipo de cómputo empleado:

No. de Lista o Brigada:

Semestre: 2020-2

Fecha de entrega: 28 De Abril de 2020

Observaciones:

CALIFICACIÓN: _____

Guía Práctica de Estudio 12: Recursividad.

OBJETIVO:

El objetivo de esta guía es aplicar el concepto de recursividad para la solución de problemas.

ACTIVIDADES:

- Revisar el concepto y las reglas de la recursividad y sus implicaciones.
- Ejecutar programas guardados en archivos desde notebook.

REPOSITORIO DE LA GUÍA:

Jupyter Notebook GitHub:
https://github.com/eegkno/FI_UNAM/blob/master/02_Estructuras_de_datos_y_algoritmos_1/P13/EDyA12.ipynb

Jupyter Notebook Visualizador:
http://nbviewer.jupyter.org/github/eegkno/FI_UNAM/blob/master/02_Estructuras_de_datos_y_algoritmos_1/P12/EDyA12_II.ipynb

Es sumamente importante respetar las indentaciones al momento de escribir código en Python. Se recomienda usar 4 espacios por nivel de indentación, los espacios son preferidos sobre el uso de tabuladores (https://www.python.org/dev/peps/pep-0008/#code-lay-out).

INTRODUCCIÓN:

Python es un lenguaje de programación **versátil multiplataforma y multiparadigma** que se destaca por su código legible y limpio. Una de las razones de su éxito es que cuenta con una licencia de código abierto que permite su utilización en cualquier escenario. Esto hace que sea uno de los **lenguajes de iniciación** de muchos programadores siendo impartido en escuelas y universidades de todo el mundo. Sumado a esto cuenta con grandes compañías que hacen de este un uso intensivo.

Recursividad

El propósito de la recursividad es dividir un problema en problemas más pequeños, de tal manera que la solución del problema se vuelva trivial. Básicamente, la recursión se puede explicar como una función que se llama así misma.

Para aplica recursión se deben de cumplir tres reglas:

- Debe de haber uno o más casos base.
- La expansión debe terminar en un caso base.
- La función se debe llamar a sí misma.

Factorial

Uno de los ejemplos más básicos es el cálculo del factorial cuya fórmula se muestra a continuación:

$$n! = \prod_{i=1}^{n} p_i = 1 \times 2 \times 3 ... \times (n-1) \times n$$

Tip: En las notebooks se pueden agregar expresiones matemáticas usando Latex (https://latex-project.org/intro.html), tal y como se muestra en la formula anterior. La fórmula se tiene que encerrar usando $$ al principio y al final de la misma.

En el siguiente ejemplo se calcula el factorial de un número de forma iterativa usando un ciclo for

```
def factorial_no_recursivo(numero):
    fact = 1
    #Se genera una lista que ve de n a 1, el -1 indica que cada iteración se resta 1 al índice.
    for i in range(numero, 1, -1):
        fact *= i    # Esto es equivalente a fact = fact * i
    return fact
```

```
factorial_no_recursivo(5)
```

```
120
```

Como se mencionó anteriormente, para resolver un problema por medio de recursividad hay que generar problemas más pequeños. Analizando la forma en que se calcula el factorial en la función pasada se tiene que:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Si se remueve el 5 se tiene:

$$4! = 4 \times 3 \times 2 \times 1 = 4(4x-1)! = 4 \times (3!)$$

Se puede afirmar que

$$4 \times 3! = 4 \times [3(3-1)!] = 4 \times 3 \times (2!)$$

Si se aplica esto a toda la secuencia, al final tenemos la siguiente expansión:

$$5! = 5(4!) = 5 \times 4 \times (3!) = 5 \times 4 \times 3 \times (2!) = 5 \times 4 \times 3 \times 2 \times (1!) = 5 \times 4 \times 3 \times 2 \times 1 \times (0!) = 120$$

Aplicando las reglas explicadas en un principio sobre recursividad, se puede resolver el problema del factorial por medio de recursión de la siguiente manera:

```
def factorial_recursivo(numero):
    if numero < 2:    #El caso base es cuando numero < 2 y la función regresa 1
        return 1
    return numero * factorial_recursivo(numero - 1) #La función se llama a sí misma
```

```
factorial_recursivo(5)
```

De la ejecución de factorial_recursivo() se puede observar lo siguiente:

- El caso base permite terminar la recursión.
- Conforme se va decrementando la variable número, se aproxima al caso base. El caso base ya no necesita recursión debido a que se convirtió en la versión más simple del problema.
- La función se llama a sí misma y toma el lugar del ciclo for usado en la función factorial_no_recursivo().
- Cada que se llama de nuevo a la función, ésta tiene la copia de las variables locales y el valor de los parámetros.

Tip: En el caso de Python, hay un límite en el número de veces que se puede llamar recursivamente una función, si se excede ese límite se genera el error: maximum recursion depth exceeded in comparison. Este límite puede ser modificado, pero no es recomendable.

Huellas de tortuga

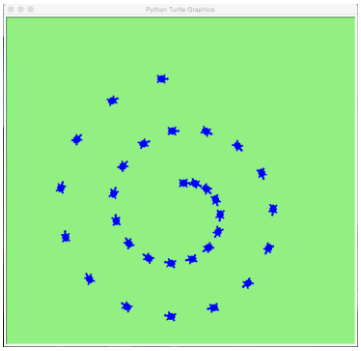
Para el siguiente ejemplo, se va a utilizar la biblioteca turtle. Como se observa en la siguiente imagen, hay una tortuga que se desplaza en espiral. Este ejemplo ha sido tomado del tutorial de la biblioteca turtle que se puede consultar en http://openbookproject.net/thinkcs/python/english3e/hello_little_turtles.html.

El objetivo es hacer que la tortuga deje un determinado número de huellas, cada una de las huellas se va a ir espaciando incrementalmente mientras ésta avanza. A contiuación se muestra la sección de código que hace el recorrido de la tortuga.

NOTA: La siguiente sección de código no se va a ejecutar en la notebook

```
#Archivo: recorrido_no_recursivo.py

for i in range(30):          #Esta determinado que se impriman 30 huellas de la
                             tortuga
tess.stamp()                 # Huella de la tortuga
size = size + 3              # Se incrementa el paso de la tortuga cada
                             iteración
tess.forward(size)           # Se mueve la tortuga hacia adelante
tess.right(24)               # y se gira a la derecha
```



¿Cómo hacer el recorrdio de la tortuga de manera recursiva? Primero se tiene que encontrar el caso base y después hacer una función que se va llame a sí misma. En esta función, el caso base es cuando se ha completado el número de huellas requerido. A continuación, se muestra el código de la función para el recorrido de la tortuga.

```
#Archivo: recorrido_recursivo.py
def recorrido_recursivo(tortuga, espacio, huellas):
     if huellas > 0:
          tortuga.stamp()
          espacio = espacio + 3
          tortuga.forward(espacio)
          tortuga.right(24)
          recorrido_recursivo(tortuga, espacio, huellas-1)
```

NOTA: El código completo de las dos versiones se encuentra guardado en los archivos recorrido_recursivo.py y recorrido_no_recursivo.py. Estos se van a ejecutar desde la notebook como si de una ventana de comandos se tratara.

Tip: En las notebooks se pueden ejecutar comandos del sistema operativo, sólo se tiene que agregar el signo de admiración antes del comando (**!comando**). Si el comando no es del sistema operativo, se despliega un aviso.

Al momento de ejecutar las siguientes instrucciones, se abre una ventana donde se muestra el desplazamiento de la tortuga. Cuando se termina de ejecutar el código, es necesario cerrar la ventana para que finalice la ejecución en la notebook.

```
#Ejecutando el código no recursivo.
!python recorrido_no_recursivo.py
```

Tip: Para la implementación recursiva (recorrido_recursivo.py) se hace uso de la biblioteca argparse, esta biblioteca permite mandar datos de entrada al programa por medio de banderas, tal y como se hace con los comandos del sistema operativo.

```
     ap = argparse.ArgumentParser()

     #El dato de entrada se ingresa con la bandera --huellas
     ap.add_argument("--huellas", required=True, help="número de huellas")

     #Lo que se obtiene es un diccionario (llave:valor) , en este caso
     llamado args
     args = vars(ap.parse_args())

     # Los valores del diccionario son cadenas por lo que se tiene que
     # transformar a un entero con la función int()
     huellas = int(args["huellas"])
```

El código se ejecuta de la siguiente manera:

```
# Como se observa, hay un espacio después del nombre del archivo
```

```
# y un espacio después de la bandera
!python recorrido_recursivo.py --huellas 25
```

La ventaja de utilizar esta forma de mandar datos de entrada al programa, es que hace la validación por nosotros, ya que si no se especifica la bandera o se especifica un valor, se genera un mensaje de error.

```
!python recorrido_recursivo.py --huella
```

Fibonacci

Recordando, la implementación iterativa para calcular la sucesión de Fibonacci es:

```
def fibonacci_iterativo_v2(numero):
    f1=0
    f2=1
    for i in range(1, numero-1):
        f1,f2=f2,f1+f2 #Asignación paralela
    return f2
```

Esta función se puede transformar a su versión recursiva de la siguiente manera:

```
def fibonacci_recursivo(numero):
    if numero == 1:     #Caso base
        return 0
    if numero == 2 or numero == 3:
        return 1
    return fibonacci_recursivo(numero-1) + fibonacci_recursivo(numero-2) #Llamada recursiva
```

```
fibonacci_recursivo(13)
```

Al igual que en la versión iterativa, se están repitiendo operaciones. Para calcular el elemento 5 se tiene:

```
f(5) =
    (n-1) = f(4)+f(3)+f(2)+f(1)
    (n-2) = f(3)+f(2)+f(1)
```

Retomando lo visto en la práctica 11, es posible mejorar la eficiencia del algoritmo si se utiliza **memorización.**

```
#Memoria inicial
memoria = {1:0, 2:1, 3:1}
```

```
def fibonacci_memo(numero):
    if numero in memoria:      #Si el número ya se encuentra calculado, se regresa el valor ya no se hacen más cálculos
        return memoria[numero]
    memoria[numero] = fibonacci_memo(numero-1) + fibonacci_memo(numero-2)
    return memoria[numero]
```

```
fibonacci_memo(13)
```

La memoria cambia después de la ejecución. En comparación con la versión iterativa de la guía 11, la función fibonacci_memo() tiene acceso a la variable memoria, por lo que efectua menos operaciones.

memoria:

```
{1: 0,
2: 1,
3: 1,
4: 2,
5: 3,
6: 5,
7: 8,
8: 13,
9: 21,
10: 34,
11: 55,
12: 89,
13: 144}
```

A diferencia de la versión anterior, como los resultados se están guardando en la variable memoria, el número de operaciones que se realizan es menor. Para calcular el elemento 5 con la nueva implementación se tiene:

```
memoria = {1:0, 2:1, 3:1}

      f(5) =
            (n-1) = f(4)+memoria(3)+memoria(2)+memoria(1)
            (n-2) = memoria(3)
```

Desventajas de la recursividad

- A veces es complejo generar la lógica para aplicar recursión.
- Hay una limitación en el número de veces que una función puede ser llamada, tanto en memoria como en tiempo de ejecución.

DESARROLLO Y ACTIVIDADES:

Código de la práctica:

Factorial_no_recursivo.py - C:\Users\Eder Berno Acosta\Documents\U.N.A.M\Facultad de Ingeniería\Estructuras de Datos y Algoritmos I\Práctica 12 EDA_Python\Factorial_no_recursivo.py (3.8.2)

File Edit Format Run Options Window Help

```
def factorial_no_recursivo(numero):
    fact = 1
    #Se genera una lista que va de la n a 1, el -1 indica que cada iteración se resta. 1 al indice
    for i in range(numero,1,-1):
        fact *=i #Esto es equivalente a fact = fact *i
    return fact
```

Factorial_recursivo.py - C:\Users\Eder Berno Acosta\Documents\U.N.A.M\Facultad de Ingeniería\Estructuras de Datos y Algoritmos I\Práctica 12 EDA_Python\Factorial_recursivo.py (3.8.2)

File Edit Format Run Options Window Help

```
def factorial_recursivo(numero):
    if numero < 2: #El caso base es cuando numero <2 y la función regresa I
        return 1
    return numero * factorial_recursivo(numero-1) #La función se llama a si misma
```

Fibonacci_recursivo.py - C:\Users\Eder Berno Acosta\Documents\U.N.A.M\Facultad de Ingeniería\Estructuras de Datos y Algoritmos I\Práctica 12 EDA_Python\Fibonacci_recursivo.py (3.8.2)

File Edit Format Run Options Window Help

```
def fibonacci_recursivo(numero):
    if numero == 1: #Caso base
        return 0
    if numero == 2 or numero == 3:
        return 1
    return fibonacci_recursivo(numero-1) + fibonacci_recursivo(numero-2) #Llamada recursiva
```

Fibonacci_memo.py - C:/Users/Eder Berno Acosta/Documents/U.N.A.M/Facultad de Ingeniería/Estructuras de Datos y Algoritmos I/Práctica 12 EDA_Python/Fibonacci_memo.py (3.8.2)

File Edit Format Run Options Window Help

```
#memoria inicial
memoria = {1:0, 2:1, 3:1}

def fibonacci_memo(numero):
    if numero in memoria: #Si el numero ya se encuentra calculado, se regresa el valor ya no se hacen más cálculos
        return memoria[numero]
    memoria[numero] = fibonacci_memo(numero-1) + fibonacci_memo(numero-2)
    return memoria[numero]
```

CONCLUSIONES

Gracias a esta práctica se pudo comprender a fondo el uso de la recursividad; así como también sus ventajas y desventajas en la aplicación para un algoritmo. Para ello se ejemplifico con condiciones simples el uso de la recursividad dentro de un algoritmo ya sea para obtener el factorial de un número o la escala Fibonacci.

BIBLIOFRAGÍA:

- Design and analysis of algorithms; Prabhakar Gupta y Manish Varshney; PHI Learning, 2012, segunda edición.
- Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein; The MIT Press; 2009, tercera edicion.
- Problem Solving with Algorithms and Data Structures using Python; Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates; 2011, segunda edition.
- http://openbookproject.net/thinkcs/python/english3e/hello_little_turtles.html