Universidad Nacional
Autónoma de México
Facultad de Ingeniería

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: 12

NOMBRE: Reyes Mendoza Miriam Guadalupe

SEMESTRE: 2020-2

FECHA DE ENTREGA: 28/04/2020

OBSERVACIONES:

CALIFICACIÓN:

RECURSIVIDAD

OBJETIVO

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

INTRODUCCIÓN

RECURSIVIDAD

Las funciones tienen la capacidad de llamarse a sí mismas, lo que se conoce como *recursividad* o *recursión*. La *programación recursiva*, hace uso de estas funciones, nos permitirá resolver problemas que de la forma habitual hasta ahora (programación iterativa o mediante bucles) sería muy complicado. Lo consigue aplicando un método en el que la solución a un problema se basa en resolver casos más pequeños el mismo problema.

Una *función recursiva* Debe cumplir una importante condición para ser utilizada en un programa: tiene que terminar. Lo logrará sí con cada llamada recursiva la solución del problema se reduce y nos acercamos hacia un caso límite, qué ocurre cuando el problema puede resolverse sin más recursiones. De no alcanzar nunca ese caso límite podríamos entrar en un bucle infinito.

FACTORIAL

Es común en matemáticas la definición de determinadas funciones dividiéndolas en dos partes: *Caso Límite* y *Caso Genérico*. En el ejemplo concreto del factorial tenemos:

0! = 1 y **n! = n*(n-1)!**

El factorial De 0 es 1; el de un número n mayor que 0 ser el mismo factorial de (n-1). En el caso genérico de la definición de factorial es donde tenemos la recursividad, ya que para calcular n! debemos calcular (n-1)!.

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

- El caso base permite terminar la recursión.
- Conforme se va decrementando la variable numero, 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.

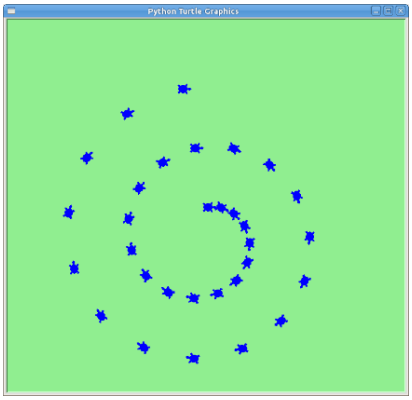
Practica 12 (Ejemplos).ipynb

Code | Download | GitHub | Binder

```
[1]: 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

[2]: factorial_no_recursivo(5)

[2]: 120

[3]: 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

[4]: factorial_recursivo(5)

[4]: 120

[5]: #Al tratar de calcular el factorial de 1000 se excede el límite permitido de recursiones
     #factorial_recursivo(1000)
```

HUELLAS DE TORTUGA

Las funciones para manejar la tortuga no son parte del lenguaje Python, sino que son provistas por el módulo ***turtle***. 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 continuación, se muestra la sección de código que hace el recorrido de la tortuga.

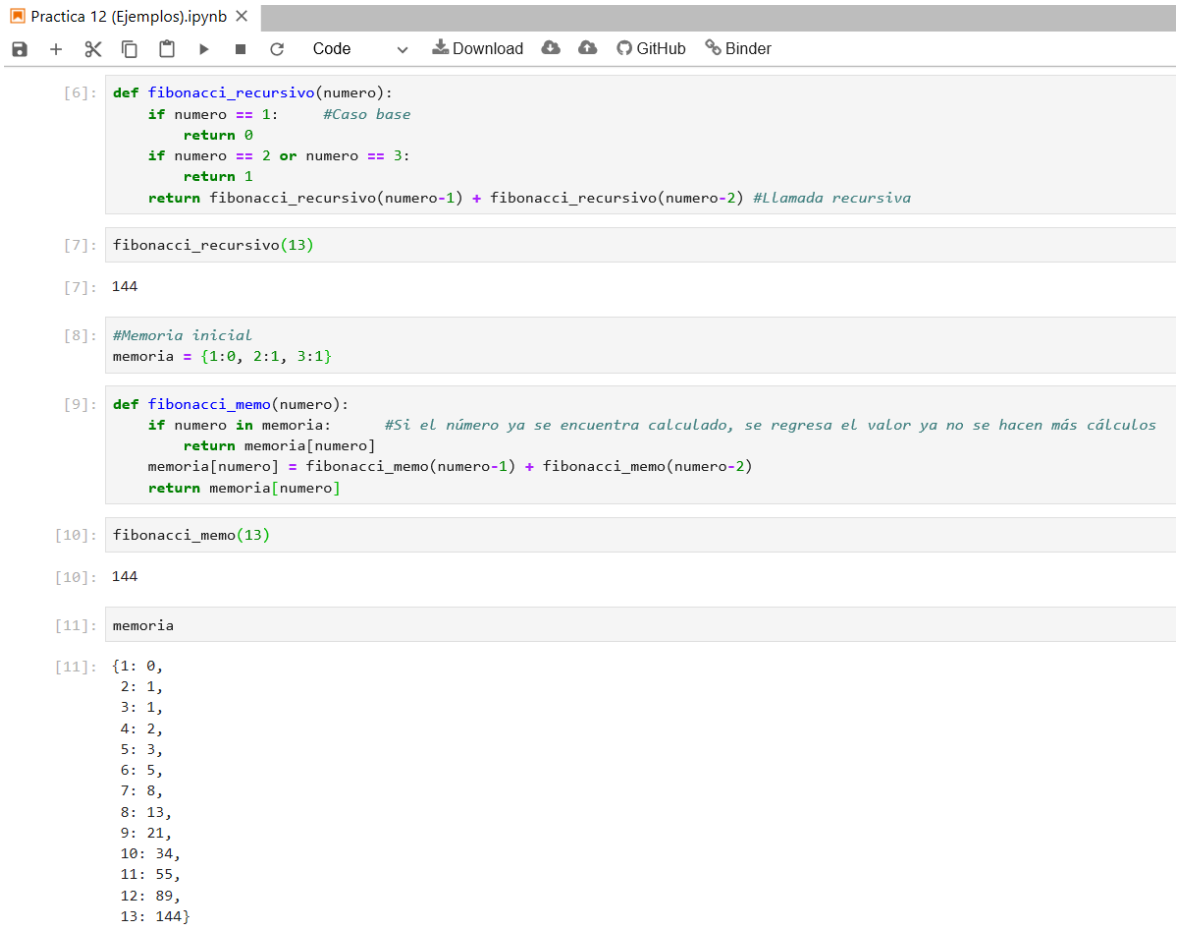


Para hacer ***recursivo***, primero se tiene que encontrar el caso base y después hacer una función que se va llame a sí misma. En esta función, el caso base es cuando se ha completado el número de huellas requerido. A continuación, se muestra el código de la función para el recorrido de la tortuga.

FIBONACCI

Es posible mejorar la eficiencia del algoritmo si se utiliza ***memorización***. 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 efectúa menos operaciones.



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 y no requiere de arreglos.

DESVENTAJAS

La principal ***ventaja*** de la recursividad frente a los algoritmos iterativos es que da lugar a ***algoritmos simples y compactos***.

La principal desventaja es que la recursividad resulta más lenta y consume más recursos al ejecutarse.
La razón por la que las funciones recursivas consumen más recursos que las iterativas es por la forma de resolverse las llamadas recursivas. Al ejecutarse una llamada función recursiva se almacena en la pila:

- Los argumentos de la función o subrutina.
- Las variables locales del subprograma.

- La dirección de retorno, es decir el punto del programa que debe ejecutarse una vez acabe la llamada actual.
- 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

Todo algoritmo recursivo puede convertirse en su equivalente iterativo y viceversa.
La recursividad será útil cuando la naturaleza de los datos así lo impongan o cuando convenga dividir un problema en casos más sencillos. Un ejemplo son los recorridos de los árboles, lo ideal para su tratamiento es utilizar algoritmos recursivos, aunque se puedan utilizar iterativos, mediante el uso de pilas.

CONCLUSIONES

Con esta práctica pude observar que existen diferentes funciones que nos van a permitir que los programas o los códigos sean muchísimo más eficientes e igualmente muchas estrategias que nos permiten realizarlo de una manera más cómoda y práctica. Y esto pudimos utilizar diferentes diseños o estrategias de algoritmos e ir viendo los pasos que implicar cada uno de ellos si es que los vamos a utilizar.

Yo creo que además de los ejercicios que vienen las prácticas para aprender necesitas ponerlos en práctica y estar buscando nuevas formas de hacer un mejor programa con herramientas que sean útiles prácticas y eficientes para el problema que queramos solucionar. Esto es lo que estamos estado viendo durante todo el curso formas de optimizar tiempo de ejecución, código y memoria, esto es realmente útil durante la práctica. A fin de cuentas, creo que como programador siempre debes estar buscando estrategias que para ti sean útiles y te permitan un buen manejo de toda la información, pero también que sean buenas y vayan mejorando los códigos cada vez más, con su dominio que se va aprendiendo con el paso del tiempo y sobre todo con la práctica.

BIBLIOGRAFÍA

- Cuevas, A. (2019). *Programar con Python 3* (1ª edición). Madrid, España: RA-MA.
- Cobo, A. (s.f.). *Programar desde un punto de vista científico*. Madrid, España: Visión Libros.
- Arboledas, D. (2017). *Criptografía sin secretos con Python* (1ª edición). Madrid, España: RA-MA.