



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:

Edgar Tista García

Asignatura:

Estructuras de Datos y Algoritmos I

Grupo:

1

No. de Práctica(s):

12

Integrante(s):

Ugalde Velasco Armando

*No. de Equipo de
cómputo empleado:*

No. de Lista:

38

Semestre:

2020-2

Fecha de entrega:

13 de mayo de 2020

Observaciones:

CALIFICACIÓN: _____

OBJETIVO

Aplicar el concepto de recursividad para la solución de problemas.

ACTIVIDAD 1: EJERCICIOS DE LA GUÍA

EJECUTA Y DESCRIBE LOS PROGRAMAS PROPUESTOS.

Factorial

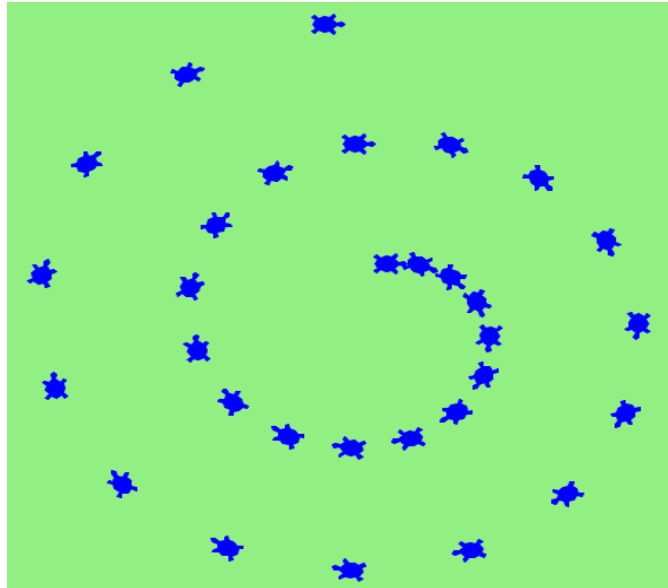
El objetivo de este programa es calcular el factorial de un número dado. La primera implementación mostrada era iterativa: se utilizó un ciclo y una variable acumuladora para calcular los productos sucesivamente, iniciando desde el número dado y finalizando con el número uno. La segunda implementación fue recursiva. Se especificó el caso base: si el número a calcular era menor a 2, entonces su factorial es 1. Después, se especificó la definición recursiva de esta expresión matemática, es decir, el factorial de un número es igual al número n multiplicado por el factorial de $n-1$. De esta forma, la función se ejecutaba recursivamente, realizando llamadas a sí misma hasta llegar al caso base, y “regresando” a las llamadas anteriores.

Huellas de tortuga

El objetivo de este programa era mostrar las “huellas de tortuga” especificadas por el recorrido en el programa. Lo anterior se realizó mediante el uso de la biblioteca **turtle**.

La primera implementación mostrada era iterativa: en cada iteración se realizaban las modificaciones o instrucciones correspondientes a cada característica del recorrido (*dirección, paso y tamaño del paso*). La segunda implementación realizada fue recursiva. El caso base de ésta consistía en que la cantidad de pasos fuera menor a 1, es decir, ya no había más pasos por dar. En cada ejecución de la función se daba un paso, realizando las modificaciones correspondientes antes mencionadas, y se realizaba una llamada recursiva a la función, reduciendo el número de pasos por uno y actualizando los nuevos argumentos. De esta forma, el número de pasos disminuía en cada llamada, hasta llegar al caso base.

Se realizó el mismo recorrido que en la versión iterativa, lo cual es un claro ejemplo de que, en la mayoría de los casos, es posible realizar implementaciones recursivas de algoritmos definidos de forma iterativa.



Recorrido de la tortuga

Fibonacci

Este programa calcula el término **n** de la serie de Fibonacci. Primero se mostró la implementación iterativa, mostrada en la práctica anterior. En ésta, se inicia con los casos base, es decir, 0 y 1, los cuales se almacenan en las variables correspondientes (f1 y f2). Posteriormente, se ejecuta un ciclo **for** el número de veces restantes para obtener el término deseado, es decir, se ejecuta **n-2** veces. Dentro de éste, se realizan las asignaciones correspondientes para calcular el siguiente término.

Después, se muestra la implementación recursiva del algoritmo. En ésta, se definen los casos base (primer y segundo términos), y se expresa la función recursivamente, acorde a su definición matemática. Como ya se mostró en la práctica anterior, esta estrategia es muy poco eficiente, ya que se realiza una gran cantidad de cálculos innecesarios, es decir, se intentan obtener términos que ya se han calculado previamente. Por lo tanto, después se muestra la implementación con memorización, la cual mejora considerablemente la eficiencia del algoritmo.

ACTIVIDAD 2: FUNCIONES RECURSIVAS

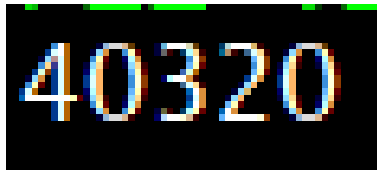
CODIFICA Y EJECUTA LAS SIGUIENTES FUNCIONES RECURSIVAS. PARA EL MANEJO MÁS CLARO Y RÁPIDO DE LAS MISMAS, PUEDES EJECUTARLAS EN CELDAS DE JUPYTER NOTEBOOK. PARA CADA FUNCIÓN, INDICA QUÉ HACE, POR QUÉ ES UNA FUNCIÓN RECURSIVA Y COMO SE COMPONE (CASO BASE, CASO RECURSIVO).

Función 1

A continuación, se muestra la implementación realizada de la función:

```
def funcion1(L):  
    if L == []:  
        return 1  
    elif len(L) == 1:  
        return L[0]  
    else:  
        return L[0] * funcion1(L[1:])  
  
lista1 = [1, 2, 3, 4, 5, 6, 7, 8]  
lista2 = funcion1(lista1)  
print(lista2)
```

Código fuente

The image shows the output of the program, which is the number 40320. The digits are displayed in a stylized, multi-colored font (blue, green, yellow, red) against a black background.

Salida del programa

El objetivo de esta función es calcular el producto de todos los elementos presentes en un arreglo. Los casos bases son:

- **Lista vacía:** se retorna el valor 1.
- **Lista de longitud 1:** se retorna el valor del único elemento presente en la lista.

Es una función recursiva, ya que en cierto punto realiza una llamada a sí misma. El caso general o recursivo está definido de la siguiente forma:

- El **producto del arreglo** es igual al valor del primer elemento y el producto del arreglo formado por los elementos originales, excluyendo el primero. Es decir, en

cada llamada recursiva se “recorta” el primer elemento del arreglo, y se utiliza éste nuevo arreglo como argumento.

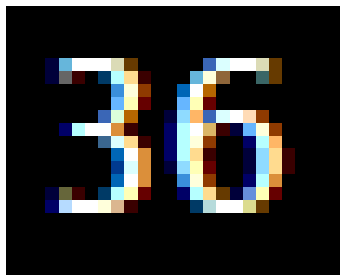
En la salida del programa logramos observar el resultado obtenido, que en efecto es el producto de todos los elementos presentes en el arreglo.

Función 3

A continuación, se muestra la implementación realizada de la función:

```
def funcion3(L,n):  
    if n == 1:  
        return L[0]  
    else:  
        return L[0] + funcion3(L[1:], n-1)  
  
lista1 = [1, 2, 3, 4, 5, 6, 7, 8]  
# Parametro corregido: se agregó n (longitud de arreglo)  
lista2 = funcion3(lista1, len(lista1))  
print(lista2)
```

Código fuente



Salida del programa

El objetivo del programa es calcular la suma de todos los elementos presentes en el arreglo. El caso base es el siguiente:

- **Lista de longitud uno:** el valor de su suma es igual al valor del único elemento presente en ella.

De no cumplirse la condición del caso base se debe tomar en cuenta el otro caso, el cual realiza una llamada recursiva a la función. Por lo tanto, es una función recursiva. Éste es el caso general o recursivo:

- El valor de la suma del arreglo es igual a la suma del primer elemento, más la suma del arreglo formado por todos los elementos, excluyendo al primero.

La definición de la función es muy clara y facilita la visualización del caso base y recursivo. Además, la función tiene como parámetro la longitud del arreglo a analizar. En cada llamada recursiva se resta una unidad del arreglo original para utilizarse como argumento, ya que, al “recortar” el primer elemento el tamaño disminuye.

La salida del programa muestra el número 36, que, en efecto, es la suma de los elementos presentes en el arreglo.

Move

A continuación, se muestra la implementación realizada de la función:

```
def move(n, x, y, z):
    if n == 1:
        print("move", x, "to", y)
    else:
        move(n-1, x, z, y)
        print("move", x, "to", y)
        move(n-1, z, y, x)

move(10, "A", "B", "C")
```

Código fuente

```
move B to A
move C to B
move A to C
move A to B
move C to B
move A to C
move B to A
move B to C
move A to C
move A to B
move C to B
move C to A
move B to A
move C to B
move A to C
move A to B
```

Fragmento de la salida del programa

El objetivo del programa es mostrar los movimientos realizados para solucionar el problema de las Torres de Hanoi, el cual consiste en mover cierta cantidad de discos de una torre de origen a una torre destino, siguiendo la regla del tamaño de los discos (no es posible colocar un disco de mayor tamaño sobre otro de menor tamaño). La función realiza llamadas a sí misma, por lo tanto, es una función recursiva. En realidad, los parámetros de la función se describen mejor de la siguiente forma:

- **x:** torre origen
- **y:** torre destino
- **z:** torre auxiliar

El caso base es el siguiente:

- **Mover el disco uno:** ya que se trata del disco más pequeño, no es necesario utilizar la torre auxiliar, cualquier movimiento es válido, por lo tanto, se coloca en la torre destino directamente.

De no cumplirse la condición del caso base se debe tomar en cuenta el caso recursivo, el cual consiste en:

1. Realizar una llamada recursiva a la función para mover los **n - 1** discos superiores de la torre **x** a la torre **z**, pero utilizando ahora como torre auxiliar a la torre **y**. Nótese que el disco más grande permanece en la torre **x**.
2. Mover el disco restante (el más grande) a la torre **y** (destino).
3. Realizar una llamada recursiva a la función para mover los **n - 1** discos de la torre **z** a la torre **y**, ahora utilizando a la torre **x** como auxiliar.

De esta forma, el programa imprime todos los movimientos a realizar para resolver el problema para el número de discos especificado.

Mystery

A continuación, se muestra la implementación realizada de la función:

```
def mystery(S):  
    N = S.split()  
    N = "".join(N)  
    if len(N) == 1 or len(N) == 0:  
        return True  
    else:  
        if N[0] == N[-1] and mystery(N[1:-1]):  
            return True  
        else:  
            return False  
  
test = input("Ingresa una cadena: ")  
print(mystery(test))
```

Código fuente

```
Ingresa una cadena: arenera  
True
```

```
Ingresa una cadena: palindromo  
False
```

Salida del programa

El objetivo del programa es determinar si la cadena introducida es un palíndromo. La función realiza llamadas a sí misma, por lo tanto, es una función recursiva.

El caso base es el siguiente:

- **Cadena de longitud cero o uno:** se devuelve el valor **True**, ya que, sin duda alguna, un solo carácter cumple con la condición de ser un palíndromo.

De no cumplirse la condición del caso base se debe tomar en cuenta el caso recursivo, el cual consiste en:

1. Comprobar si el primer carácter de la cadena es igual al último carácter.
2. Comprobar si la cadena interior es un palíndromo, realizando una llamada recursiva a la función. La cadena interior es igual a la cadena original, excluyendo el primer y último caracteres.

3. Si estas dos condiciones se cumplen, entonces la cadena analizada se trata de un palíndromo. De no cumplirse alguna, entonces la cadena no es un palíndromo.

De esta forma, el programa determina si la cadena analizada es un palíndromo o no lo es. Como se puede observar en la salida del programa, se ingresaron dos palabras y se obtuvieron los resultados esperados.

CONCLUSIONES

Gracias a la realización de la práctica logré comprender y aplicar el concepto de recursividad en los problemas presentados. La recursividad es uno de los conceptos más importantes y fundamentales en las Matemáticas y en las Ciencias de la Computación: nos permite expresar problemas de una forma muy explícita y expresiva, lo cual, sin duda alguna, facilita su resolución en un gran número de casos.

La recursión también puede considerarse una alternativa a la iteración, ya que es posible implementar algoritmos iterativos en forma recursiva. Algunos casos en los que se presentó la situación anterior son los ejemplos mostrados en la guía de laboratorio y en la práctica 11.

Los ejercicios realizados en la práctica fueron idóneos para analizar y comprender algunos problemas o algoritmos definidos en forma recursiva. En algunos casos fue complicado identificar el objetivo de los programas, ya que los identificadores no tenían nombres claros respecto a su función en el algoritmo: he ahí la dificultad de los ejercicios.

En mi opinión, la recursividad es una forma de atacar los problemas extremadamente útil, principalmente por su “poder expresivo”. Sin embargo, puede no facilitar la legibilidad de los algoritmos o programas en todos los casos, o bien, no proporcionar una solución eficiente al problema. Por ejemplo, la implementación recursiva del algoritmo para calcular el n -ésimo término de la serie de Fibonacci tiene un rendimiento pésimo (sin considerar la utilización de memorización), a diferencia de la implementación iterativa. Cabe mencionar que, en cambio, su legibilidad es excelente, ya que el código es muy similar a la definición de la función original.

Es necesario analizar en cada caso la estrategia de implementación a utilizar, de tal forma que facilite el entendimiento de los algoritmos y, principalmente, sea eficiente.