Taller de Introducción a la programación: Practica recursividad

M. Sc. Saúl Calderón Ramírez
Instituto Tecnológico de Costa Rica,
Escuela de Computación.

24 de marzo de 2017

Resumen

Los siguientes son un conjunto de ejercicios recursivos resueltos, tomados del libro en construcción por el Máster Jeff Schmidt Peralta, coordinador de los cursos de introducción a la programación del área de Ing. en computadores, del ITCR.

1. Determinar sin un número tiene ceros (recursividad simple)

El problema a resolver es determinar si un número entero dado, tiene entre sus dígitos al menos un cero. El comportamiento de la ejecución debe ser como se muestra a continuación

```
>>> ceros(3214)
False
>>> ceros(14098)
True
>>> ceros(0)
True
```

Se definen en primer términos las entradas, salidasy restricciones. En este caso la salida es un valor de verdad.

Entradas: número

Salidas: indicador booleano de si el número contiene al menos un cero

Restricciones: número entero.

El algoritmo se basa en descomponer el número de entrada, por medio de las funciones de obtener el último dígito y eliminar el último dígito. Cada dígito es analizado y si existe un cero, se debe terminar retornando un valor de verdadero. En caso que todos los dígitos sean revisados y no exista un cero, se retorna un valor de cero. El código en Python de esta solución es:

```
# Función: ceros
# Entradas: número
# Salidas: indicador boolean si el número contiene un cero
# Restricciones: número debe ser entero

def ceros(num):
    if isinstance(num, int):
        if num != 0
            return ceros_aux(abs(num))
        else:
            return True
    else:
        return "Error: numero debe ser entero"

def ceros_aux(num):
    if num == 0:              # condición de terminación 1
        return False          # valor de retorno
    elif (num % 10) == 0:     # condición de terminación 2
        return True           # valor de retorno
    else:
        return ceros_aux(num // 10)
```

En este caso se está utilizando el concepto de función principal y auxiliar que ya se había mostrado. En la función principal se evalúa que el número de entrada debe ser entero, así como el caso especial de que el número sea cero, que en la función recursiva va a servir como condición de terminación. Es importante notar que esta función tiene dos condiciones de terminación:

• cuando el número se convierte en cero (producto de ir eliminando sus dígitos), en cuyo caso se retorna un valor de falso (False).

• cuando alguno de los dígitos es cero, para retornar verdadero (True).

A continuación se muestran ejemplos de la ejecució no corrida corrida corrida de la función:

```
>>> ceros(820414)
ceros_aux(82041)
ceros_aux(8204)
ceros_aux(820)
True
```

2. Formar un número con los dígitos pares de otro (recursividad de pila)

Se desea construir una función que forme un número a partir de otro, considerando solo los dígitos pares del número de entrada. Esta función debe comportarse de la siguiente forma:

```
>>> pares(3214)
24
```

Como parte de la comprensión del problema, definimos las entradas, salidas y restricciones del mismo.

Entradas: número

Salidas: número con los dígitos pares del número de entrada

Restricciones: número entero

El algoritmo se basa en descomponer el número de entrada, por medio de las funciones de obtener el último dígito y eliminar el último dígito. Cada dígito es analizado y si es par va a formar parte de la salida del problema, creando un nuevo número.

Para construir un nuevo número vamos a considerar que cada dígito en un número es una potencia de 10. Supongamos que tenemos los dígitos 3, 5, 8, 6 (de menos significativo a más significativo) y a partir de ellos vamos a formar el número 6853. Analicemos la siguiente secuencia de operaciones:

```
3 * 1 = 3
3 + 5 * 10 = 53
53 + 8 * 100 = 853
853 + 6 * 1000 = 6853
```

Se utiliza un factor de multiplicación, que es una potencia de 10, para irle dando su correspondiente valor al dígito. El valor inicial del factor es 1 (proviene de 10 0), debido a que el primer dígito (menos significativo) del número resultante multiplicado por 1 da como resultado ese dígito. Por cada dígito siguiente, el factor, se va multiplicando por 10, es decir los factores van a ser 1, 10, 100, 1000, 10000 y así sucesivamente. Note que cada factor es una potencia de 10. El resultado de cada producto se suma al siguiente producto, para obtener al final el número que se desea.

En este caso se utiliza el concepto de función principal para la restricción y auxiliar para hacer la recursividad. Además se está agregando un argumento al realizar la llamada a la función auxiliar (factor de multiplicación), que se necesita para hacer los cálculos. La solución encontrada sigue siendo de recursividad de pila, ya que la llamada recursiva es parte de una operación, en este caso, la suma delos productos. En forma manual, se puede probar la solución, siguiendo la secuencia de llamadas y operaciones:

```
>>> pares(92614)
pares_aux(92614, 1)
4 * 1 + pares_aux(9261, 10)
4 * 1 + pares_aux(926, 10)
4 * 1 + 6 * 10 + pares_aux(92, 100)
4 * 1 + 6 * 10 + 2 * 100 + pares_aux(9, 1000)
4 * 1 + 6 * 10 + 2 * 100 + pares_aux(0, 10000)
4 * 1 + 6 * 10 + 2 * 100 + 0
4 + 60 + 200 + 0
264
```

3. Número de apariciones de un dígito en un número (recursividad de pila)

Se desea construir una función que reciba un dígito y un número entero y cuente las veces que aparece el dígito en el número. El comportamiento de la ejecución debe ser como se muestra a continuación:

```
>>> apariciones(4, 34214)
2
>>> apariciones(7, 140981)
0
```

Inicialmente se definen las entradas, salidas y restricciones.
Entradas: dígito, número
Salidas: cantidad de veces que aparece el dígito en el número
Restricciones: dígito sea válido, número entero

El proceso consiste en obtener cada dígito del número y compararlo con el dígito dado como entrada. En caso que sean iguales se van contando para mostrar el resultado final y si son diferentes se continúa con la revisión.

Por medio de condiciones se verifica cada una de las restricciones del problema, que el dígito sea válido y además que el número sea entero. Efectuando una prueba manual de la función, podemos encontrar la siguiente secuencia de llamadas y resultados:

```
>>> apariciones(4, 34214)
apariciones_aux(4, 34214)
1 + apariciones_aux(4, 3421)
1 + apariciones_aux(4, 342)
1 + apariciones_aux(4, 34)
1 + 1 + apariciones_aux(4, 3)
1 + 1 + apariciones_aux(4, 0)
1 + 1 + 0
2
```

4. Factorial de un número (recursividad de pila)

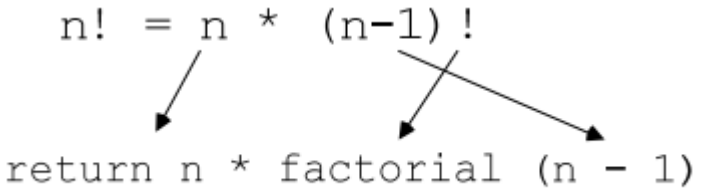
El problema de cálculo del factorial de un número entero es un ejemplo típico de recursividad. El factorial un número se define de la forma siguiente:

$$n! = \begin{cases} 1 & \text{si}\,, n = 0 \\ n \cdot (n-1)! & \text{si}\,, n > 0 \end{cases}$$

```
0! = 1
1! = 1 * 0! = 1 * 1 = 1
2! = 2 * 1! = 2 * 1 * 0! = 2 * 1 * 1 = 2
3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 *1 = 6
.
.
.
n! = n * (n-1)!
```

La solución consiste en aplicar la fórmula recursiva, usando como entrada el valor del factorial que se desea calcular. Las entradas, salidas y restricciones se van a indicar únicamente en el código.

En este caso puede verse la relación entre la definición del factorial y la llamada recursiva en el código de la función:



Se presenta una ejecución manual de la función factorial:

```
>>> factorial(6)
factorial_aux(6)
6 * factorial_aux(5)
6 * 5 * factorial_aux(4)
6 * 5 * 4 * factorial_aux(3)
6 * 5 * 4 * 3 * factorial_aux(2)
6 * 5 * 4 * 3 * 2 * factorial_aux(1)
6 * 5 * 4 * 3 * 2 * 1 * factorial_aux(0)
6 * 5 * 4 * 3 * 2 * 1 * 1
720
```

Puede observarse la pila de llamadas, en la cual sevan acumulando los productos, que se resuelven cuando se encuentra la condición de terminación.

5. La sucesión de Fibonacci (recursividad de pila)

En matemáticas, la sucesión de Fibonacci *combinatoria* es la siguiente sucesión infinita de números naturales:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,.... , a partir de su definición:

$$f(n) = \begin{cases} 1 & \text{si }, n = 0 \\ 1 & \text{si }, n = 1 \\ f(n-1) + f(n-2) & \text{si }, n > 1 \end{cases}$$

La definición original define el caso trivial de $n = 0$ como $f(0) = 0$:

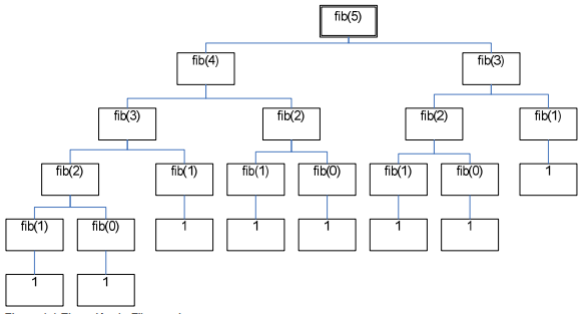
$$f(n) = \begin{cases} 0 & \text{si }, n = 0 \\ 1 & \text{si }, n = 1 \\ f(n-1) + f(n-2) & \text{si }, n > 1 \end{cases}$$

A cada elemento de esta sucesión se le llama númerode Fibonacci. Esta sucesión fue descrita en Europa por Leonardo de Pisa, matemático italiano del siglo XIII también conocido como Fibonacci.

El primer elemento es 1, el segundo es 1 y cada elemento restante es la suma de los dos anteriores. El tercer término de la sucesión se obtiene sumando el segundo y el primero, el cuarto, a partir de la suma del tercero y el segundo y así sucesivamente. El problema a resolver sería calcular el valor del n-ésimo término de la sucesión, que se obtendrá sumando los términos $n-1$ y $n-2$.

```
fib(0) = 1
fib(1) = 1
.
.
.
fib(n) = fib(n - 1) + fib(n - 2) , para n = 2,3,4,5,...
```

En esta función es de suma importancia notar que se realiza una doble llamada recursiva con diferentes argumentos cada vez que se ejecuta la función. La ejecución de esta función tendría una forma similar a la siguiente:



6. Elevar un número a una potencia (recursividad de pila)

Una función muy utilizada es elevar un número a una potencia o exponenciación. Aún cuando vimos que Python provee funciones para realizarla, es importante comprender como la función puede ser programada. Elevar un número a una potencia significa multiplicar un número (base) por sí mismo cierta cantidad de veces (exponente). La base puede ser cual cualquier número real y la potencia se va a considerar como un número entero mayor a igual a cero. La exponenciación descrita anteriormente puede escribirse matemáticamente mediante la fórmula:

$$x^0 = 1$$

.
.
.

$$x^n = x * x^{(n-1)} \quad \text{para cualquier n entero y mayor que cero}$$

Puede notarse que existe un proceso repetitivo en la fórmula, que puede modelarse como un proceso recursivo. La recursividad consistiría en ir multiplicando el número la cantidad de veces que indique el exponente.

Puede notarse que la llamada recursiva es un reflejo exacto de la fórmula de la exponenciación. Se presenta una ejecución manual dela función elevar:

```
>>> elevar(2, 5)
elevar_aux(2, 5)
2 * elevar_aux(2, 4)
2 * 2 * elevar_aux(2, 3)
2 * 2 * 2 * elevar_aux(2, 2)
2 * 2 * 2 * 2 * elevar_aux(2, 1)
2 * 2 * 2 * 2 * 2 * elevar_aux(2, 0)
2 * 2 * 2 * 2 * 2 * 1
32
```

Al encontrar la condición de terminación se resuelven los productos (de la base), considerando que por definición de la fórmula:

$$x^0 = 1$$

7. Calculo de una sumatoria de números naturales predecesores(recursividad de pila)

Las sumatorias son fórmulas utilizadas para describir el comportamiento de los términos, desde un límite inferior hasta un límite superior. La fórmula se describe utilizando la letra griega sigma-mayúscula

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + ... + n$$

La variable i describe el índice de la sumatoria, que inicia en el límite inferior y va a llegar hasta n, que es el límite superior. En el cuerpo de la sumatoria, se describe el comportamiento que tendrá cada término a sumar. En esta caso, cada término a sumar es i, desde que vale 0 hasta que llega al límite superior n.

La ejecución de la función muestra el siguiente comportamiento:

```
>>> sumatoria_i(5)
sumatoria_i_aux(5)
5 + sumatoria_i_aux(4)
5 + 4 + sumatoria_i_aux(3)
5 + 4 + 3 + sumatoria_i_aux(2)
5 + 4 + 3 + 2 + sumatoria_i_aux(1)
5 + 4 + 3 + 2 + 1 + sumatoria_i_aux(0)
5 + 4 + 3 + 2 + 1 + 0
15
```

8. Calculo de la sumatoria de un intervalo

Se desea construir una función que sume desde un valor inicial denominado A hasta un valor final denominado B, donde A y B son dos números enteros. Así mismo A debe ser menor o igual a B. Se debe implementar la fórmula matemática descrita por:

$$\sum_{i=A}^{B} i = A + (A + 1) + (A+ 2) + (A + 3) + ... + B$$

El código de la función indicando sus entradas, salidas y restricciones sería: # Función: sumatoria de un intervalo # Entradas: a, b (límites de sumatoria) # Salidas: sumatoria según fórmula dada # Restricciones: n >= 0, a <= b

```
>>> sumatoria_int(10, 15)
sumatoria_int_aux(10, 15)
10 + sumatoria_int_aux(11, 15)
10 + 11 + sumatoria_int_aux(12, 15)
10 + 11 + 12 + sumatoria_int_aux(13, 15)
10 + 11 + 12 + 13 + sumatoria_int_aux(14, 15)
10 + 11 + 12 + 13 + 14 + sumatoria_int_aux(15, 15)
10 + 11 + 12 + 13 + 14 + 15
75
```

9. Algoritmo de Euclides para encontrar el MCD de un número

El máximo común divisor (mcd) es el mayor entero que divide con residuo de cero o en forma exacta a dos números enteros. Existen varios métodos para

implementar la función de *mcd*. A continuación se presenta un método llamado el algoritmo de Euclides. Este método utiliza residuos sucesivos entre los números. Por ejemplo, tenemos que si se desea calcular:

```
mcd(12, 16)
= (mcd 16 12)  # el residuo entre 12 y 16 es 12
= (mcd 12  4)  # el residuo entre 16 y 12 es 4
= (mcd  4  0)  # el residuo entre 12 y 4 es 0.
= 4            # Termina algoritmo.
```

Se van calculando los residuos, tomando para la siguiente repetición el número que sirvió como divisor y el residuo. Cuando el residuo es cero, se toma el último divisor como máximo común divisor. Implemente una función *calcularMCD(numero1, numero2)* que retorne el máximo común divisor entre *numero1* y *numero2*.

10. Determinar si un número es primo

El problema de determinar si un número es primo tiene diversas soluciones y ha sido ampliamente estudiado por diversos matemáticos. Un número es primo si y solo si es divisible en forma exacta únicamente por 1 y por sí mismo. La función a construir debe recibir un número entero y retornar un valor booleano que indique si el número es primo o no es primo. El proceso de solución es un proceso repetitivo, que va a buscar entre los números entre 2 y el mismo número (sin considerarlo), para encontrar si alguno lo divide en forma exacta. Un número divide en forma exacta a otro si el residuo de la división entera es cero, es decir:

```
si num1 % num2 == 0
```

implica que *num1* es divisible entre *num2*, lo cual es lo mismo que *num2* es un divisor exacto de *num1*. Implemente la función *verificarSiEsPrimo(numeroPrimo)* la cual retorne verdadero si el número recibido es primo.

11. Determinar el número de combinaciones de un conjunto

Sea un conjunto de n elementos. Las combinaciones de m elementos del conjunto son subconjuntos de m elementos (sin importar el orden). El número de combinaciones indica la cantidad de formas en que se pueden extraer subconjuntos a partir de un conjunto dado. Supongamos que se tiene un conjunto con 6 objetos diferentes A, B, C, D, E, F, de los cuales se desea escoger 2 (sin importar el orden). Existen 15 formas de efectuar la elección:

A,B	A,C	A,D	A,E	A,F
B,C	B,D	B,E	B,F	C,D
C,E	C,F	D,E	D,F	E,F

Recursivamente, se puede definir el número de combinaciones de m objetos tomados de n, denotado (n, m) por:

$$f(n,m) = \begin{cases} 1 & \text{si } m = 0 \text{ o } m = n \\ f(n-1,m) + f(n-1,m-1) & \text{sino} \end{cases}$$

```
(n, m) = 1                          si m = 0 ó m = n
(n, m) = (n - 1, m) + (n - 1, m - 1)    en otro caso
```

Puede notarse que esta solución, al igual que Fibonacci, utiliza **recursividad doble**. La ejecución muestra los siguientes resultados:

```
>>> ncsr(7, 4)
35
>>> ncsr(5, 3)
10
>>> ncsr(6, 2)
15
```

Escriba la función *evaluarCombinatoria(n,m)* la cual retorne la cantidad de combinaciones posible al tomar *m* elementos de un conjunto de *n*.

12. Implementación recursiva de la multiplicación

La operación de multiplicar dos números a * b, cuando b es un número entero, puede calcularse recursivamente por medio de b - 1 sumas sucesivas del número a. Por ejemplo la multiplicación 8 * 4, podría verse como:

$$8 * 4 = 8 + 8 + 8 + 8$$

Escriba una función recursiva *multi(num1, num2)* que reciba dos números e implemente la multiplicación por medio de sumas sucesivas

13. Invertir un entero recursivamente

Escriba una función *invertir(num)* utilizando recursividad de pila que reciba un número entero e invierta sus dígitos. El comportamiento de la función debe ser como se indica a continuación:

```
>>> invertir(-4592)
-2954

>>> invertir(50000)
5

>>> invertir(50001)
10005
```

14. Sumar recursivamente los dígitos de un número

Escriba una función recursiva *suma_dig(num)* que reciba un número que puede ser entero o real y sume sus dígitos. La función debe brindar resultados como los que se presentan en los siguientes ejemplos:

```
>>> suma_dig(4123)
10

>>> suma_dig(10001)
2

>>> suma_dig(122.45)
14
```

15. Detector recursivo de palíndromos en números

Un número es palíndromo si puede leerse igual de izquierda a derecha que de derecha a izquierda. Escribir una función llamada *palindromo(num)*que reciba un número y retorne True si el número recibido es un palíndromo y False si no. Por ejemplo:

```
>>> palindromo(38583)
True

>>> palindromo(2442)
True

>>> palindromo(4)
True

>>> palindromo(1010010)
False
```

16. Reproducción de insectos

Si de 5 parejas de hormigas cada una engendra 3 hormiguitas, luego mueren, dejando que las 15 hormiguitas nacidas, engendren también 3 hormiguitas por pareja, luego mueren y se sigue repitiendo el proceso. Escriba una función recursiva hormiguitas que recibe como argumento un entero positivo y que determine el número de hormiguitas qué existirán al cabo de n períodos. Considere que siempre se pueden formar las parejas. Algunos resultados de la función se muestran a continuación:

```
>>> hormiguitas(0)
15

>>> hormiguitas(1)
21

>>> hormiguitas(4)
66
```

17. Extraer el dígito mayor

Escriba una función recursiva de pila *extraerDigitoMayor(numero)* que reciba un número entero y obtenga el dígito mayor. La función debe comportarse como los siguientes ejemplos:

```
>>> extraerDigitoMayor(1029099);
9
>>> extraerDigitoMayor(810260);
8
>>> extraerDigitoMayor ("ein_esel_lese_nie");
"Entrada inválida"
```

18. Concatenación de números

Escribir una función *num_append(num1, num2)* que encadene los dígitos del segundo número con el primer número, de acuerdo al comportamiento siguiente:

```
>>> num_append(124, 56)
12456

>>> num_append(2340, 0)
23400
```

19. División truncada por dígito

Escribir una función en Python *div(dig, num)* que reciba como argumentos un dígito entre uno y nueve, y un número entero. La función debe retornar el número dividido (truncando) en cada uno de sus dígitos.

```
>>> div(2, 4625)
2312

>>> div(1, 4625)
4625
>>> div(4, 87250)
21010
```

20. Tupla con digitos mayores y menores que 5

Escriba una función digitosque reciba un número (que debe ser entero) y retorne una tupla que tenga la siguiente forma: (cantidad-dígitos-mayores-que-5,

cantidad-dígitos-menores-o-iguales-que-5) La función debe retornar su resultado en forma similar al siguiente ejemplo:

```
>>> digitos(482401)
(1, 5)

>>> digitos(4)
(0, 1)
```

21. Corrimiento de un entero

Construya una función en Python llamada *shift(num)* que reciba un número entero y mueva cada dígito una posición hacia adelante, el dígito menos significativo lo pone como dígito más significativo. Por ejemplo:

```
>>> shift(4321)
1432

>>> shift(10000)
1000

>>> shift(100001)
110000
```

22. Prueba de digitos de un número contenidos en otro número

Escriba una función *booleana dig_ab(num1, num2)* que recibe 2 números enteros A, B y retorna True si todos los dígitos deel primer número están contenidos en el segundo número y False en otro caso. La función debe retornar resultados como los ejemplos siguientes:

```
>>> dig_ab(3, 3333)
True

>>> dig_ab(123, 632)
False
```

23. Multiplicación de los digitos

Escriba una función *multiplicarDigitos(num1, num2)* que recibe 2 números enteros del mismo tamaño y forma un nuevo número con la multiplicación de cada dígito del primer número con cada dígito del segundo número. Si la multiplicación de 2 dígitos es mayor a 9, se toma el dígito menos representativo del resultado de la multiplicación. Los siguientes ejemplos muestran como debe comportarse la ejecución de la función:

```
>>> multdig(24, 42)
88

>>> multdig(323, 388)
964

>>> multdig(153, 632)
656
```

24. Cantidad de números primos en intervalo

Escriba una función *calcularCantidadPrimosEnIntervalo(a, b)* que reciba dos números enteros *a* y *b* y determine la cantidad de números primos que existe entre esos dos números (incluyéndolos). El resultado debe ser como se muestra a continuación:

```
>>> num_primos(1, 7)
5

>>> num_primos(1, 16)
7
```

25. Desplazamiento de un número

Construya una función en Python llamada *desplazar(num)* que reciba un número entero y mueva cada dígito una posición hacia adelante, el dígito menos significativo lo pone como dígito más significativo. Por ejemplo:

```
>>> shift(4321)
1432

>>> shift(10000)
1000

>>> shift(100001)
110000
```