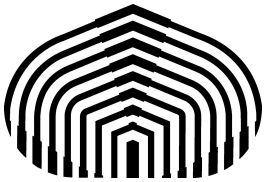

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la información
CI-2691- Laboratorio de algoritmos I

Laboratorio 8 (Parte II)

El objetivo de este laboratorio es dar una introducción a recursión, soluciones recursivas, recursión de cola y sus eficiencias.

Recursión

Como se ha explicado en los laboratorios anteriores, un problema complicado puede ser dividido en uno o más subproblemas. Realizar tal división corresponde a hacer un *análisis descendente* del problema en cuestión. Cuando el análisis descendente de un problema incluye la resolución de otra instancia de ese mismo problema, diremos que el análisis propone una solución *recursiva.* Muchos problemas pueden resolverse usando recursión. En particular, si un problema puede dividirse en una o más instancias del mismo problema, pero con una entrada más pequeña, diremos que la solución propuesta utiliza la técnica: *divide y conquistaras.*

.

Recurrencias

Muchas funciones y propiedades son enunciadas en formas de *recurrencias*. Por ejemplo, a continuación se muestra la recurrencia que define la serie de Fibonacci.

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Soluciones Recursivas

La recursión no sólo puede utilizarse para calcular el valor de funciones, sino también para resolver problemas. En particular, la resolución de un problema puede depender de la resolución de instancias más pequeñas de ese problema.

Por ejemplo:

- Consideremos que tenemos un arreglo de enteros, ordenados de menor a mayor.
- Queremos decidir si un elemento **elem** se encuentra en el arreglo o no.

Una posible solución sería recorrer el arreglo y comparar cada elemento con **elem.** Sin embargo, podemos formular una solución alternativa:

- Se revisa un elemento aleatorio **x** del arreglo y se compara con **elem**
- Si **x == elem** entonces reportamos que el elemento se encuentra en el arreglo.
- De lo contrario, pudieran ocurrir dos casos:

    ✓ Si **x < elem** debemos reformular la búsqueda. Sin embargo, cualquier elemento anterior a **x** es menor o igual que **x** (ya que supusimos que el arreglo estaba ordenado). Por lo tanto, cualquier elemento anterior a **x** está estrictamente menor que **elem.** Siendo así, podemos restringir nuestra búsqueda solamente a los elementos que estén después de **x.**

    ✓ Si **x > elem** podemos utilizar un argumento análogo al anterior, para justificar que la búsqueda se realice únicamente sobre los elementos que estén antes de **x.**

Notemos que el arreglo que se pasa como entrada a los subproblemas es siempre más pequeño que el original. Eventualmente, el arreglo será de tamaño cero. En tal caso, ya no tiene sentido dividir el problema. Podemos simplemente reportar que el elemento no se encuentra en el arreglo (no hay elemento alguno en un arreglo vacío).

Podemos formular nuestra solución recursiva como una recurrencia, de la siguiente manera:

$$esta(A, elem, i, j) = \begin{cases} false & i >= j \\ true & A[i] = elem \\ esta(A, elem, k+1, j) & A[i] < elem, i <= k < j \\ esta(A, elem, i, k) & A[i] > elem, i <= k < j \end{cases}$$

Note que se toma una k arbitraria, tal que i ≤ k < j . Resultado en investigación han demostrado que, en la mayoría de los casos, la mejor escogencia para tal k es el valor (i+j)/2. Esto implica que el elemento a comparar del arreglo, siempre será aquel que se encuentre en la mitad de la porción considerada del mismo. A la estrategia anterior, junto con la escogencia propuesta para k, se le conoce como el algoritmo de *búsqueda binaria*.

Este algoritmo es mucho más eficiente que la propuesta original de recorrer el arreglo completo. La razón por la que es tan eficiente será discutida más adelante, en el curso de Algoritmos y Estructura II

Recursión Indirecta

Es posible que un problema tenga una solución que sea recursiva, pero de manera indirecta. Por ejemplo, un subprograma *p1* que llame a otro subprograma *p2*, cuando a su vez *p2* llama a *p1*. Tal clase de recursión es conocida como *recursión indirecta*.

Existen muchos ejemplos concretos de recursión indirecta que se tratarán en cursos futuros. Sin embargo, a efectos de este laboratorio solo trataremos con subprogramas directamente recursivos. Esto no nos restringe en poder de expresión, ya que cualquier conjunto de subprogramas que sea indirectamente recursivo, puede re-escribirse como un solo subprograma directamente recursivo.

Iteración vs. Recursión

Una propiedad fundamental en la computación es que todo problema que tenga una solución iterativa, también tendrá una recursiva. Así mismo, todo problema que tenga una solución recursiva, también tendrá una iterativa. Escoger qué solución implementar dependerá de muchos factores, como la eficiencia, la facilidad de codificación, la legibilidad, entre otros.

Eficiencia de la Iteración

Generalmente, una solución iterativa será más eficiente que una recursiva (la razón de esto será mucho más clara adelante, en el curso de Organización y Arquitectura del Computador). Una solución iterativa no debe invocarse a sí misma, lo cual tendría un costo en tiempo y espacio. Sin embargo, diseñar soluciones iterativas no siempre es sencillo. Muchos problemas tienen soluciones que son naturalmente recursivas. Un buen ejemplo es el cálculo de la serie de Fibonacci. La solución recursiva es inmediata de la definición, sin embargo es sumamente ineficiente. La versión iterativa, aunque mucho más eficiente, es más difícil de diseñar.

Recursión de Cola

La recursión es tan natural y común, que muchos lenguajes han adoptado una optimización especial conocida como *Recursión de Cola*. La optimización consta en transformar automáticamente un subprograma recursivo por una versión iterativa, siempre y cuando tal subprograma cumpla con una característica particular: cada llamada recursiva no debe ser

sucedida por acción alguna. **Hay que tener en cuenta que Python no optimiza la recursión de cola.**

Consideremos la recurrencia siguiente, que define el factorial de un entero:

$$fact(n) = \begin{cases} 1 & n = 0 \\ n * fact(n-1) & n > 0 \end{cases}$$

Al implementar tal solución directamente, la llamada recursiva a fact sería sucedida por la multiplicación con n. Por lo tanto, tal implementación no sería recursiva de cola. Consideremos ahora, la recurrencia en (4), que define el factorial de una manera alternativa, usando un acumulador como ayuda:

$$fact(n) = fact\_aux(n, 1)$$

$$fact\_aux(n, acum) = \begin{cases} acum & n = 0 \\ fact(n-1, acum * n) & n > 0 \end{cases}$$

Si implementamos directamente esta nueva solución, podremos notar que la función recursiva fact_aux si es recursiva de cola. Una vez se realiza la llamada recursiva, no hay que realizar acción alguna adicional.

Terminación de una Recursión

Así como en una iteración se debe plantear una cota (con la finalidad de asegurar que tal iteración termina), en un subprograma recursivo se debe plantear igualmente una cota. Esto, ya que se corre el mismo peligro de no terminar. Particularmente, si las llamadas recursivas no se hacen sobre instancias más pequeñas que la original, entonces el programa puede recurrir sin fin.

De ésta manera, todo subprograma recursivo debe incluir una función de cota que:

- Debe depender únicamente de los parámetros de entrada
- Debe ser siempre no-negativa
- En cada llamada recursiva, debe decrecer (la función de cota del subprograma llamado, debe ser estrictamente menor que la del llamador).

Ejercicios:

1. (PreLab8Ejercicio2.gcl y PreLab8Ejercicio2.py):Escriba una función recursiva en GCL que dado un número N devuelva el valor de la sumatoria $(\sum i : 0 <= i < N : 2^i)$. Para calcular el valor de la potencia utilice, también una función recursiva. Escriba un algoritmo en GCL que permita calcular dicha sumatoria para diferentes valores de N. Traduzca su algoritmo a un programa en Pythom.

2. (PreLab8Ejercicio3.py): Escriba un programa, que por cada línea leída desde un archivo, determine cuántas de sus palabras son palíndromos. Para ello, escriba una función recursiva que determine si una palabra es palíndromo. Una palabra es palíndromo si se puede leer de igual manera a pesar de ser invertida. Por ejemplo: ana, arepera y abba.

Guarde sus programas con los nombres sugeridos y súbalos en el aula virtual

Referencias:

[1]. Intput and Output - Python Programming. Disponible en la Web.
http://en.wikibooks.org/wiki/Python_Programming/Input_and_Output