A library of recursive functions

Ernesto P. Adorio

February 6, 2004

1 Recursion

A recursive function is one which calls itself in the body of the function. A characteristic of a recursive function is the elegance and conciseness of the formulation or implementation. Some programming languages depend on recursion as the major way of performing computations. The (slow) execution speed and the (high) memory requirements of recursive functions however usually makes one look for an iterative version (if there is!) of the function.

In writing recursive functions, the implementer must ensure that the function can return in finite time for reasonable low enough input argument values. There must not be runaway or infinite recursion.

In the following section, we describe some well-known recursive functions which return a number. It is to be understood that the implementation is not meant for production use but for academic purposes only. We have not even added a maximum recursion depth level test for safety. This is left as an exercise for the reader or might be incorporated in future versions. Several computer science textbooks discuss ways to remove tail recursion, a case where the self-calls are at the end of the function body.

2 Recurf.py, a Python module of recursive functions

Our recursive functions are grouped in the `recurf.py` module and most find application in the field of combinatorics. We encourage the user to determine the recursion formula used from the Python sources and even to modify the sources to print the trace and level of recursion.

- fibo(n:integer): integer

  fibo(n) computes the nth number in the sequence $1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

- lucas(n: integer): integer

  lucas(n) is related to fibo with the diference only in the starting values. The first 10 lucas numbers are $1, 3, 4, 7, 11, 18, 29, 47$.

- factorial(n: integer): integer

  factorial(n) returns the factorial of n, denoted as n!, and defined as $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$.

  It can be interpreted as the total number of ways of permuting a set of n objects. For a deck of cards, the number of ways of dealing out in order the 52 cards is the value 52! =

  $$80658175170943878571660636856403766975289505440883277824000000000000L$$

- ncr(n, r): integer

  ncr(n, r) returns the number of combinations (disregarding order) of n things taken r at a time. The formula is given by

  $$nCr = \binom{n}{r} = \frac{n!}{(n-r)!r!}$$

  . These numbers also appear as the binomial coefficients in the expansion of $(x+y)^n$.

  Let $n = 5$. The list of the binomial coefficients for $k = 0, \ldots, n$ is given by

  ```
  >>> [ recurf.ncr(5, k) for k in range(6) ]
  [1, 5, 10, 10, 5, 1]
  >>>
  ```

- npr(n,r):integer

  npr(n, r) returns the number of permutations of n things taken r at a time. The formula is given by

  $$nPr = \frac{n}{(n-r)!}$$

- ackermann(m, n : integer):

  This function is highly recursive and is well studied in computer science. The function exhibits highly explosive growth and glacially slow runs even for small m and n. Try making a list of all ackermann numbers for $m, n \leq 4$:).

- catalan1(n:integer): integer

- catalan2(n:integer): integer

  The function catalan1() and catalan2() employs two different recursive ways to compute the nth Catalan number. The nth Catalan number can be interpreted in various ways as (url:http://mathforum.org/advanced/robertd/catalan.html)

1. the number of ways a polygon with $n + 2$ sides can be cut into$n$ triangles
2. the number of ways in which parentheses can be placed in a sequence of numbers to be multiplied, two at a time.
3. the number of planar binary trees trees with $n + 1$ leaves
4. the number of paths of length 2n through an n-by-n grid that do not rise above the main diagonal.

A non-recursive formula for computing the Catalan number is given by $\frac{1}{n+1}\binom{2n}{n}$. The list of the first 20 Catalan numbers:

```
>>> [ recurf.Catalan(i) for i in range(20) ]
[0, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796,
58786, 208012, 742900, 2674440, 9694845L, 35357670L,
129644790L, 477638700L, 1767263190L]
>>>
```

- stirling1(n, k: integer): integer

  The stirling number of the first kind can be computed by the recursive function stirling1(n, k). The number of permutations of n elements which contain k cycles is given by $(-1)^{n-k}Stirling1(n, k)$.

- stirling2(n, k): integer

  The stirling number of the second kind can be interpreted as the number of ways of partitioning a set of n objects into k sets or blocks.

- bell(n): integer

  The nth Bell number is the total number of ways of partitioning a set of n objects into $1, 2, \ldots, n$ sets or blocks.

3 Source codes

Python source codes for computing recursive functions is in the module `recurf.py` whose source is shown in the following listing.

```
"""
recurf.py  - recursive routines in python

This module is for academic purposes only.
Version 0.1.0  Feb. 6 web release.
Author: Ernesto P. Adorio
"""
```

```
def fibo(n):
    """Returns the nth fibonacci number using recursion."""
    if n < 2:
        return 1
    return fibo(n-1) + fibo(n-2)

def lucas(n):
    """Returns the nth lucas number using recursion"""
    if n < 2:
        return 1
    if n == 2:
        return 3
    return lucas(n-1) + lucas(n-2)

def factorial(n):
    """Returns the nth factorial number using recursion."""
    if n <= 1: return 1
    return n * factorial(n - 1)

def ncr(n,r):
    """Returns the value of nCr = n! / [(n-r)! r!] using recursion. """
    if n < 2 or r == 0 or r >= n: return 1
    return ncr(n-1, r) + ncr(n-1, r-1)

def npr(n,r):
    """Returns the value of nPr = n! / [ (n -r)! ] using recursion. """
    if r == 0: return 0
    if r == 1: return n
    return npr(n, r - 1) * (n - r + 1)

def ackermann(m, n):
    """Returns the ackermann function using recursion. (m, n are non-negative)."""
    if m == 0 and n >= 0: return n + 1
    if n == 0 and m >= 1: return ackermann(m-1, 1)
    return ackermann(m-1, ackermann(m, n-1))

def catalan1(n):
    """Returns the nth Catalan number using recursion."""
    if n <= 2: return n
    return (2*n)*(2*n-1) * catalan1(n-1) / ((n +1)*n)

def catalan2(n):
    """Returns the nth Catalan number using a second slower (very very slooow) recursion
    if n < 2: return 1

    sum = 0L
```

```
    for i in range(1, n):
        sum = sum + catalan2(i) * catalan2(n-i)
    return sum

def stirling1(n,k):
    """Returns the stirling number Stirl2(n,k) of the second kind using recursion."""
    print "n = " , n, "k = ", k
    if k <= 1 or k == n: return 1
    if k > n or n <= 0: return 0
    return stirling1(n-1, k-1) + n * stirling1(n-1, k)

def stirling2(n,k):
    """Returns the stirling number Stirl2(n,k) of the second kind using recursion."""
    print "n = " , n, "k = ", k
    if k <= 1 or k == n: return 1
    if k > n or n <= 0: return 0
    return stirling2(n-1, k-1) + k * stirling2(n-1, k)


def bell(n):
    """Returns the nth Bell number using recursion."""
    if n < 2: return 1
    sum = 0
    for i in range(1, n+1):
        sum = sum + ncr(n-1, k-i) * bell(n-k)
    return sum
```