5 Iteration and Recursion

1. **(Review)** Consider computing the summation from 1 to n.

   (a) Confirm that the following program using for works.

```
def sum_loop(n)
  s = 0
  print "sum=", s, "\n"
  for i in 1..n do
     s = s + i
     print "sum=", s, "\n"
  end
  s
end
```

   (b) Confirm that the following program using recursion works.

```
def sum(n)
  print "Compute sum(",n,")...\n"
  if n >= 2
    print "sum(", n, ")=sum(", n-1 ,")+", n, "\n"
    s = sum(n-1) + n
  else
    s = 1
  end
  print "sum(", n, ")=", s, "\n"
  s
end
```

2. Consider the function mult_sum(p,n) that computes the sum of multiples of p between 1 and n ($p \leq n$).

   (a) Make the function using repetition.

   (b) Make the function using recursion.

3. **(Using repetition)**. Define the following functions using repetition.

   (a) a function factorial_loop(n) that computes the factorial of n, that is, the product of all positive integers less than or equal to n. (factorial_loop.rb)

   (b) a function power2_loop(n) that computes 2^n. Do not use **. (power_loop.rb)

   (c) a function power_loop(x, n) that computes x^n. Do not use **. (power_loop.rb)

   (d) a function taylor_e_loop(x, n) that computes the following series

$$\sum_{k=0}^{n} \frac{\mathsf{x}^k}{k!}.$$

   Note that this is the Taylor series of e^x when $n \to \infty$. (taylor_e_loop.rb)

4. **(Using recursion)**. Define the following functions using recursion.
   (a) a function factorial(n) that computes the factorial of n, that is, the product of all positive integers less than or equal to n. (factorial.rb)
   (b) a function power2(n) that computes 2^n. Do not use **. (power.rb)
   Hint: 2^n is equal to $2 \times 2^{n-1}$.
   (c) a function power(x, n) that computes x^n. Do not use **. (power.rb)
   (d) a function taylor_e(x, n) that computes the following series
   $$\sum_{k=0}^{n} \frac{\mathsf{x}^k}{k!}.$$
   Note that this is the Taylor series of e^x when $n \to \infty$. (taylor_e_loop.rb)

5. **(Using repetition)**. Define the following functions using repetition. (prime_loop.rb)
   (a) a function nod_loop(k,n) that computes the number of divisors of k among all positive integers less than or equal to n.
   (b) a function nop_loop(n) that computes the number of prime numbers among all positive integers less than or equal to n.
   (c) a function msod_loop(n) that computes the maximum sum of divisors, that is, the integer k in 1,...,n such that the sum of divisors sod(k,k) is maximized.

6. **(Using recursion)**. Define the following functions using recursion. (prime.rb)
   (a) a function nod(k,n) that computes the number of divisors of k among all positive integers less than or equal to n.
   (b) a function nop(n) that computes the number of prime numbers among all positive integers less than or equal to n.
   (c) a function msod(n) that computes the maximum sum of divisors.
   Hint: Let sod(k,k)=s_k. Then this problem is equivalent to finding the maximum of $s_1, \ldots, s_n$. The maximum of $s_1, \ldots, s_n$ is equal to "the maximum of the maximum of $s_1, \ldots, s_{n-1}$" and s_n. That is, we have the following relation.
   $$\mathsf{msod}(n) = \begin{cases} \mathsf{msod}(n-1) & \text{if } \mathsf{msod}(n-1) \geq s_n \\ s_n & \text{if } \mathsf{msod}(n-1) < s_n \\ s_1 & \text{if } n = 1. \end{cases}$$

7. **(Using repetition)**. Define the following functions using repetition.
   (a) Define a function np_loop(n) that computes the next prime number (the minimum prime greater than or equal to n). If n is a prime, then np_loop(n)=n. (prime_loop.rb)
   (b) Define a function nth_prime_loop(p,n) that computes the nth prime number greater than the prime p. For example, nth_prime_loop(5,3) is 13. (prime_loop.rb)

(c) Define a function tnpo(n) that returns the half of n if n is even, and 3n+1 if n is odd. Mathematician Collatz conjectured[1] that every integer n comes to 1 by applying tnpo repeatedly. For example, if we have 3, then we obtain $3 \Rightarrow 10 \Rightarrow 5 \Rightarrow 16 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$ by applying tnpo repeatedly.

Here, we define a function collatz (n) that computes the number of repetition times that we apply tnpo to come to 1. For example, collatz(16)=4, collatz(5)=5, and collatz(3)=7. Using the repetition, make a Ruby function collatz_loop (n) to compute collatz (n). (collatz_loop.rb)

8. **(Using recursion)**. Define the following functions using recursion not using repetition.

   (a) Define a function np(n) that computes the next prime number (the minimum prime greater than or equal to n). If n is a prime, then np(n)=n, and if n is not a prime, then it is equal to the minimum prime which is at least n+1. (prime.rb)

   (b) Define a function nth_prime(p,n) that computes the nth prime number greater than the prime p. For example, nth_prime(5,3) is 13. In fact, nth_prime(5,3) is equal to nth_prime(7,2), because the next prime is 7. Since the next prime is 11, it is equal to nth_prime(11,1), and therefore it is equal to the next prime 13. (prime.rb)

   (c) Make a function collatz (n) defined in the previous problem.
   Hint: Consider the relationship between collatz (n) and collatz (tnpo(n)). (collatz.rb)

9. The operation to concatenate two arrays is represented by +. For example, [0]+[0]=[0,0] and [1,2]+[3,4]=[1,2,3,4].

   (a) Re-define the function make1d(s) in a recursive way.

   (b) Re-define the function make2d(h,w) in a recursive way.

10. **(Making an array)** Define the following functions to make an array with dimension at least three.

    (a) Define a function make3d(a,b,c) that makes an a×b×c array all of whose entries are 0, based on the definition of make2d(a,b,c). (make3d.rb)

    (b) Define a function makend(n,m) that makes a m× ··· ×m(n times) array. (makend.rb)
    Hint: Consider a recursive definition.

    (c) Define a function makearray(d) that makes an array whose size is defined by an array d. For example, makearray([2,4,3]) makes a $2 \times 4 \times 3$ array. (makearray.rb)

[1] this problem is an open problem in mathematics, that is, there are no proof and no counterexample.

5.1 Applications

11. **(combination number by recursion)** The combination number ${}_nC_k$ is the number of combinations of choosing k items out of n items.

    (a) Using the fact
    $$ {}_nC_k = \frac{n!}{k!(n-k)!}, $$
    explain why it holds that
    $$ {}_nC_k = \begin{cases} 0 & \text{if } k > n \\ 1 & \text{if } k = 0 \\ {}_{n-1}C_{k-1} + {}_{n-1}C_k & \text{otherwise.} \end{cases} $$

    (b) Define a function combination(n,k) that computes the combination number in a recurive way.

12. **(combination number by repetition)** Using FOR-loop, make a function combination_loop(n,k) that computes ${}_nC_k$. Compare combination(n,k) and combination_loop(n,k) increasing n and k, and discuss which is faster.

13. **(Sierpinski Triangle)**. Define a function sierpinski(n) that makes a 2-dimensional array with size n× n such that the (i, j) entry is equal to the remainder of ${}_iC_j$ when divided by 2. (if $i < j$ then the value is 0) Use the function show to display the obtained array in isrb. The results will be depicted as in Figure 1, where black and white are reversed for visibility.

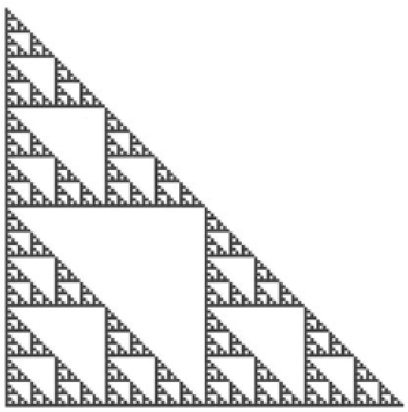

Figure 1: Sierpinski Triangle

14. **(Tower of Hanoi)**. The goal of the game "Tower of Hanoi" is to move all the disks from the left peg to the middle one. Only one disk may be moved at a time. A disk can be placed either on an empty peg or on top of a larger disk. Try to move all the disks using the smallest number of moves possible.

(a) Let Hanoi(n, a, b, c) be a function that describes a procedure to move n disks from the left beg a to the middle peg b using the right peg c. Complete the program in Slides.

(b) Let Hanoi_times(n, a, b, c) be a function computes the minimum number of times of moving disks when we do Hanoi(n, a, b, c). Complete the program in Slides.

(c) Compute Hanoi(4,"a","b","c"), Hanoi(5,"a","b","c"), Hanoi(6,"a","b","c"), and Hanoi(50,"a","b","c").

15. Make a program that displays the following arrow, in which we can determine the size(height) of the arrow.

```
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
         *
         *
         *
         *
         *
         *
         *
         *
         *
         *
```

16. We want to define a function that converts a number n to binary. Make such a function in two ways: using repetition and recursion.

17. (a) Consider finding the maximum value in a given array a. Define such a function maxElem(a,i) using recursion, where it returns the maximum value of a[0],. . . , a[i]. If i=0, return a[0]. Otherwise, if `maxElem(a,i-1) < a[i]` then return a[i], and otherwise return maxElem(a,i-1). That is, the maximum value of the array a can be obtained by maxElem(a,n-1).

(Example: maxElem([4,3,5,1],3)=5.)

(b) Consider finding the second largest value of a given array a. Define a function secondMaxElem(a) using recursion.

Hint: Let max be the maximum value between the first i-1 elements in a, and max2 be the second largest one. Define secondMaxElem2(a,max,max2,i) that returns the second largest value in a[0],..., a[i]. This can be used to define secondMaxElem(a). (We may suppose that all the entries in the array a are distinct.)