Recursion

Recursion

- An algorithm is called recursive when it builds upon itself

- Single recursion
  - For example, defining the function x^n
    - Functions are defined recursively

$$x^n = \begin{cases} 1 \; khi \; n = 0 \\ x \times x^{n-1} \; khi \; n \geq 1 \end{cases}$$

    - Recursive algorithm

```
function power (x, n)
begin
 if (n=0) then power = 1
 else ham_mu = x * power(x, n-1)
 endif
end
```

Recursion

- Multiple recursion
  - A recursive definition can have more than one recursive call
  - For example
    - Definition of Fibonacci sequence

      $F_0 = 1, F_1 = 1$

      $F_n = F_{n-1} + F_{n-2}$
    - Algorithm

```
function fib(n)
begin
 if ((n=0) or (n=1)) then fib = 1
 else fib = fib(n-1) + fib(n-2)
 endif
end
```

Recursion

- Mutual recursion
  - Definitions are said to be mutual recursion, if they depend on each other
  - For example
    - Definition of even/odd numbers

$$even(n) = \begin{cases} true\ if\ n = 0 \\ odd(n-1)\ else \end{cases} \qquad odd(n) = \begin{cases} false\ if\ n = 0 \\ even(n-1)\ else \end{cases}$$

    - Algorithm

```
function even (n)
begin
 if (n=0) then even = true
 else even = odd(n-1)
 endif
end
```

```
function odd (n)
begin
 if (n=0) then odd = false
 else odd = even(n-1)
 endif
end
```

Recursion

- Nested recursion
  - Definitions are called recursively nested
  - For example
    - Definition of Ackermann function

$$A(m,n)=\begin{cases} n+1 \ if \ m=0 \\ A(m-1,1) \ if \ m>0, n=0 \\ A(m-1, A(m,n-1)) \ else \end{cases}$$

    - Algorithm

```
function (m, n)
begin
 if (m=0) then Ackermann = n+1
 else if (n=0) Ackermann = Ackermann(m-1,1)
       else Ackermann = Ackermann(m-1, Ackermann(m, n-1))
       endif
 endif
end
```

Design of recursive algorithm

- Principle
  - We need to have
    - some cases where the solution is determined – "simple cases": the stopping cases of the recursion
    - a way to move from a "complex case" to a "simple case"
- Difficulty
  - It is guaranteed that the recursion will stop when it reaches a known solution
    - Functions must be defined across the entire data domain
- Solution
  - The sequence of consecutive values of the called parameters must vary monotonically and reach a value for which the corresponding solution has been determined

Design of recursive algorithm

- Example 1
  - The following algorithm checks whether a is a divisor of b

```
function divisor (a, b) // assume a>0, b>0
begin
 if (a ≥ b) then
  if (a=b) divisor = true
  else divisor = false
  endif
 else divisor=divisor(a, b-a)
 endif
end
```

- The sequence of values b, b-a, b-2a… continuously decreases until a≥b then it will stop, the case has been determined

Design of recursive algorithm

- Example 2
  - Algorithm

```
function syracuse (n)
begin
 if (n=0 or n=1) then syracuse = 1
 else
  if (n mod 2 = 0) syracuse = syracuse(n/2)
  else syracuse = syracuse(3*n+1)
  endif
 endif
end
```

- Clearly defined algorithms
- Does the algorithm stop?

Halting Problem

- Unable to determine halting (1)
  - Problem
    - Is it possible to build a tool that automatically checks whether an algorithm P can stop when executing on a dataset D?
    - Inputs
      - Algorithm P
      - Data set D
    - Outputs
      - true, if algorithm P stops on data set D
      - false, otherwise

Halting Problem

- Unable to determine halting (2)
  - Suppose there exists a program *terminate* that automatically checks the termination of an algorithm
  - We build the following program Q basing on *terminate*

```
program Q
begin
result = terminate(Q)
 while (result = true)
wait(1 minute)
 endwhile
end
```

The stopping problem is indeterministic!

Order of recursive calls

- Give the results of the following two algorithms

```
T(n) // n ≥0
begin
   if (n=0) then do nothing
   else
     T(n-1)
     print(n) //print n
  endif
end
```

```
G(n) // n ≥0
begin
   if (n=0) then do nothing
   else
     print(n) //print n
     G(n-1)
  endif
end
```

- Recursive algorithm to print the corresponding binary sequence of an integer

Hanoi Tower (1)

- There are 3 piles A, B and C. Each pile can stack discs of different sizes, according to the rule that larger disks must be below smaller ones. The task is to move n discs on pile A to pile C with the following conditions:
  - Only one disc can be transferred at a time.
  - There is never a situation where a large disc is stacked on top of a small disc.
  - Use one pile as an intermediate pile when moving disc.

Hanoi Tower (2)

- Assume
  - We solved the problem with n-1 disks
- Principle
  - To move n disks from pile A to pile C, do
    - Move n-1 smaller disks from pile A to pile B
    - Move the largest disk from pile A to pile C
    - Move n-1 smaller disks from pile B to pile C
- Algorithm

```
Hanoi(n, A, B, C)
begin
   if (n=1) then move the large disk from pile A to pile C
   else   Hanoi(n-1, A, C, B)
          Move the large disk from pile A to pile C
          Hanoi(n-1, B, A, C)
  endif
end
```

Hanoi Tower (3)

- Evaluating complexity
  - Calculating the number of disk moves

$$C(n) = \begin{cases} 1 & if\ n = 1 \\ C(n-1) + 1 + C(n-1) & else \end{cases}$$

$$C(n) = \begin{cases} 1 & if\ n = 1 \\ 2C(n-1) + 1 & else \end{cases}$$

$C(n) = 2^n - 1$

Recursion cost

- Function call cost

```
…
f(x, y, z)
…
```

```
f(a, b, c)
…
end
```

1. Save context

2. Create a stack to store the parameters, return address, …

3. Go to function

4. Get from stack

5. Implementation

6. Create a stack containing the returned result

7. Return to function call location

8. Get result from stack

9. Restore context

Recursion cost

- Example

```
factorial(n)
begin
 factorial = 1
 for i = 2 to n
   factorial = factorial * i
 endfor
end
```

n-1 multiplication
n assignment
n assignment (loop)
n-1 increment by 1 (loop)
n comparisons

```
factorial(n)
begin
 if (n = 1) then
   factorial = 1
 else
   factorial = n * factorial(n-1)
 endif
end
```

n-1 multiplication
n assignment
n-1 subtraction (calculate n-1)
n comparisons
n function calls

➡ Recursion cost is large due to function calls

Recursion elimination

- Converting a recursive algorithm into an equivalent algorithm that does not contain recursive calls
  - Using loops

- Two cases of single recursion
  - Tail recursion
    - An algorithm is said tail recursion if it does not contain any processing after the recursive call.
  - Non-tail recursion
    - An algorithm is said to be non-tail recursion if it contains processing after the recursive call.

Recursion elimination

- Tail recursion
  - General diagram of tail recursion

```
P(U)
begin
   if C then
      D
      P(α(U))
   else
      T
   endif
end
```

```
P'(U)
begin
   while C do
      D
      U = α(U)
   endwhile
   T
end
```

U: list of parameters
C: condition depending on U
D: processing of the algorithm
α(U): represents the parameter transformation
T: stop processing

Recursion elimination

- Tail recursion
  - Eliminate the recursion of the following algorithm

```
bsearch(X, A, l, r)
begin
   if (l ≤ r ) then
      m = (l+r)/2
      if (X = A[m]) then bsearch = m; return
      else if (X < A[m]) then bsearch = bsearch(X, A, l, m-1)
             else bsearch = bsearch(X, A, m+1, r)
             endif
      endif
   else
      bsearch = 0
   endif
end
```

Recursion elimination

- Tail recursion
  - Equivalent iterative algorithm

```
bsearch' (X, A)
begin
   l = 1
   r = n
   while (l ≤ r ) do
            m = (l+r)/2
            if (X = A[m]) then bsearch' = m; return
            else      if (X < A[m]) then r = m-1
                      else l = m+1
                      endif
            endif
   endwhile
   bsearch' = 0
end
```

Recursion elimination

- Non-tail recursion
  - It is important to memorize *the context* of the recursive call
    - Typically the *parameters* of a recursive call
  - Use stack structure *to* store context
    - Stack operations
      - create
      - isempty
      - push
      - pop
      - top
  - Two ways to eliminate non-tail recursion

Recursion elimination

- Non-tail recursion
  - Method 1

```
Q(U)
begin
   if C(U) then
      B(U)
      Q(α(U))
      E(U)
   else
      T(U)
  endif
end
```

```
Q'(U)
begin
   create(S)
   while C(U) do
      B(U)
      push(S, U)
      U = α(U)
   endwhile
   T(U)
   while not isempty(S) do
      U = top(S)
      E(U)
      pop(S)
   endwhile
end
```

Recursion elimination

- Non-tail recursion
  - Method 1
    - Illustration

Call $Q(U_0)$
$C(U_0)$ is correct
$B(U_0)$
Let $Q(\alpha(U_0))$
$C(\alpha(U_0))$ is correct
$B(\alpha(U_0))$
Let $Q(\alpha(\alpha(U_0)))$
$C(\alpha(\alpha(U_0)))$ is incorrect
$T(\alpha(\alpha(U_0)))$
$E(\alpha(U_0))$
$E(U_0)$

$\Rightarrow$

Call $Q'(U_0)$?

Recursion elimination

- Non-tail recursion
  - Method 1
    - Example

```
T(n) // n ≥0
begin
   if (n=0) then do nothing
   else
           T(n-1)
           print(n) //print n
  endif
end
```

```
T'(n) // n ≥0
begin
   create(S)
   if (n=0) then do nothing
   else
      while (n>0) do
           push(S, n)
           n = n-1
      endwhile
      while (not isempty(S)) do
           n = top(S)
           print(n) //print n
           pop(S)
      endwhile
   endif
end
```

Recursion elimina

- Non-tail recursion
  - Method 2

```
Q(U)
begin
   if C(U) then
          B(U)
          Q( α (U))
          E(U)
   else
          T(U)
   endif
end
```

```
Q'(U)
begin
   create(S)
   push(S, (newcall, U))
   while not isempty(S) do
          (state, V) = top(S)
          pop(S)
          if (state = newcall) then
             U = V
             if C(U) then
                B(U)
                push(S, (end, U))
                push(S, (newcall, α(U)))
             else T(U)
             endif
          endif
          if (state = end) then
             U = V
             E(U)
          endif
   endwhile
end
```

Recursion elimination

- Non-tail recursion
  - Method 2
    - Illustration

Call $Q(U_0)$
$C(U_0)$ is correct
$B(U_0)$
Let $Q(\alpha(U_0))$
$C(\alpha(U_0))$ is correct
$B(\alpha(U_0))$
Let $Q(\alpha(\alpha(U_0)))$
$C(\alpha(\alpha(U_0)))$ is incorrect
$T(\alpha(\alpha(U_0)))$
$E(\alpha(U_0))$
$E(U_0)$

⇒

Call $Q'(U_0)$?

Recursion elimination

- Non-tail recursion
  - Method 2
    - Example

```
T(n) // n ≥0
begin
   if (n=0) then do nothing
   else
         T(n-1)
         print(n) //print n
  endif
end
```

```
T'(n)
begin
   create(S)
   push(S, (newcall, n))
   while not isempty(S) do
         (state, k) = top(S)
         pop(S)
         if (state = newcall) then
             if (k>0) then
                   push(S, (end, k))
                   push(S, (newcall, k-1))
             else do nothing
             endif
         endif
         if (state = end) then
             print(k)
         endif
   endwhile
end
```

Recursion elimination

- Iterative algorithms are often more efficient
- Recursive algorithms are often easier to build

- Most compilers can automatically eliminate tail recursion

- It is always possible to eliminate the recursion of an algorithm

Exercises

- Problem 1
  - Definition of Fibonacci sequence

    $Fib_0 = 1$, $Fib_1 = 1$

    $Fib_n = Fib_{n-1} + Fib_{n-2}$
  - Tasks
    1. Build a recursive algorithm to calculate Fib(n)
    2. Prove that the complexity (in terms of number of additions) of the algorithm is $\Omega(2^{n/2})$
    3. Build an algorithm calculating the pair (Fib(n), Fib(n-1)) with n>0
    4. Use the algorithm in question 3 to build a new algorithm to calculate Fib(n)
    5. Evaluate the complexity (by number of additions) of the above algorithm

Exercises

- Problem 2

  The greatest common divisor of two positive integers is defined as follows
  - if $x = y$ then $usc(x, y) = x$
  - if $x > y$ then $usc(x, y) = usc(x-y, y)$
  - if $x < y$ then $usc(x, y) = usc(x, y-x)$

  1. Build a recursive algorithm to calculate the greatest common divisor of two positive integers
  2. Eliminate the recursion

- Problem 3
  1. Build a recursive algorithm to print the corresponding binary sequence of an integer
  2. Eliminate the recursion