Recursion – the mirrors

1. Introduction

- Recursion is a powerful problem solving technique
  - It breaks a problem into smaller identical problems
  - Then breaks new problem into even smaller problems
  - Eventually, new problem is small and can be solve easily (base case)
  - The solutions to small problems can lead to the solution of original problem

- Recursive functions are functions that call themselves

- Note: Uncompleted function calls are saved in the process stack!

- Example 1: [Binary Search] An array list[n] consists of n ≥ 1 distinct integers such that list[0] ≤ list[1] ≤ ... ≤ list[n-1].

  Goal: We want to figure out if an integer "searchNum" is in the list. If it is, we should return an index, i, such that list[i] = searchNum; otherwise return -1.
Algorithm Outline:

// divide and conquer strategy

let "left" and "right" denote the left and right ends of the list to be searched.

initially, left = 0 and right = n-1

int search(left, right, searchNum)

    if (left > right)
        index = -1
    else
        let middle = (left + right)/2
        if (list[middle] = searchNum) index = middle
        else if (list[middle] > searchNum)
            index = search(left, middle-1, searchNum)
        else // list[middle] < searchNum
            index = search(middle+1, right, searchNum)
    return index

Example:  1  5  9  12  15  21  29  31  searchNum = 9

index:  0  1  2  3  4  5  6  7

[0,7] → [0,2] → [2,2]  // found list[2] = 9

return 2 → return 2 → return 2  // exit function!
• Example 2: Compute factorial of n

i.e. \( n! = n \times (n-1)! = n \times (n-1) \times (n-2) \times \ldots \times 1 \), for \( n > 0 \) and \( 0! = 1 \)

```cpp
#include <iostream>
int fact (int n);
using namespace std;

main()
{
    int n;
    cout << "enter a number : " << endl;
    cin >> n;
    cout << n << " factorial is " << fact(n) << endl;
}

int fact(int n)
{
    if (n==0) return(1); //base case
    else return (n * fact(n-1));
}
```
5. Recursion

Program flow: // uncompleted tasks are saved in a process stack!

assume \( n = 3 \) →

call fact(3), push return(3*fact(2)) onto stack

/* 3*fact(2) undefined */

call fact(2), push return(2*fact(1)) onto stack

/* 2*fact(2) undefined */

call fact(1), push return(1*fact(0)) onto stack

/* 1*fact(0) undefined */

call fact(0), return(1)

pop return(1*fact(0)) from stack → return (1)

pop return(2*fact(1)) from stack → return (2)

pop return(3*fact(2)) from stack → return (6)
Example 3: Tower of Hanoi

There are 3 poles and n disks of different diameter placed on the 1st pole. The disks are in order of decreasing diameter as one scan up the pole. Monks were repeatedly supposed to move disks from pole 1 to pole 3 using spare pole 2 and obeying the following rules:

(i) only one disk can be moved at any time,
(ii) no disk can be placed on the top of a disk with smaller diameter.

Major steps: // poles: source, spare & destination

Base case: n=1

move the disk from source to destination directly

Recursion: n > 1

move (n-1) disks from source to spare
move nth disk from source to destination
move (n-1) disk from spare to destination

Note: n = 1 or 2 are easy!
n = 3 disks

- Main: Move 3 disks from #1 to #3, spare #2
  
  (a) move 2 disks from #1 to #2
  (b) move 1 disk from #1 to #3
  (c) move 2 disks from #2 to #3

- From (a), source #1, destination #2, spare #3
  
  (a.1) move 1 disk from #1 to #3
  (a.2) move 1 disk from #1 to #2
  (a.3) move 1 disk from #3 to #2

- From (b), source #1, destination #3 (base case)

- From (c), source #2, destination #3, spare #1
  
  (c.1) move 1 disk from #2 to #1
  (c.2) move 1 disk from #2 to #3
  (c.3) move 1 disk from #1 to #3

n = 4 disks

- Main: Move 4 disks from #1 to #3, spare #2
  
  (a) move 3 disks from #1 to #2  // same case as n=3 disks
  (b) move 1 disk from #1 to #3  // base case
  (c) move 3 disks from #2 to #3  // same case as n=3 disks
Towers of Hanoi Program

/* n = # of disks, i = source, j = destination, k = spare */

void tower(int n, int i, int j, int k);

main()
{
    int n;
    cout << "enter a number n : " << endl;
    cin >> n;
    tower(n,1,3,2);
}

void tower(int n, int i, int j, int k)
{
    if (n==1) {  // base case
        cout    << "move top disk from pole “ << i 
               << “to pole “ << j << endl;
    } else {  // for n > 1
        tower(n-1,i,k,j);
        tower(1,i,j,k);
        tower(n-1,k,j,i);
    }
}
5. Recursion

- Problems of recursive algorithms
  - usually, recursive algorithms are simple and short
  - many recursive algorithms, such as Example 2 computing factorial, are inefficient due to the overhead associated with function calls

  In this case, should consider using simple iterative algorithm. e.g. using simple loop to solve the factorial problem.

- some recursive algorithms are inherently inefficient, i.e. they take quite sometime to terminate.

Example: Compute the number of combination of \( k \) things of \( n \),

\[
\begin{align*}
c(n,k) &= c(n-1,k-1) + c(n-1,k) & \text{for } 0 < k < n \\
c(n,n) &= c(n,0) = 1 & \text{// base case} \\
c(n,k) &= 0 \text{ if } k > n & \text{// base case}
\end{align*}
\]

In this case, should try to use other techniques to solve the problem (note: some problems may not have good solution!)

CSC510 will introduce many different techniques
2. Recognition Algorithms for Languages

- Definition: A language is a set of strings of symbols.

- Example: English, set of algebraic expressions and programming languages.

- A grammar states the rules of a language, i.e., a string can be constructed by using these rules if and only if it is in a language.

- An algorithm that determines whether a given string is in the language is called a recognition algorithm for the language.

- The recognition algorithm is usually designed, based on the grammar. Many recognition algorithms are recursive algorithms.

- Basic notations:
  - x|y means x or y
  - xy or x.y means x followed by y
  - <word> means any instance of word that the definition defines.

- Example 1: C++ identifiers

\[
\begin{align*}
  \text{<identifier>} & = \text{<letter>} | \text{<identifier><letter>} | \text{<identifier><digit>} \\
  \text{<letter>} & = a | b | .. | z | A | B | .. | Z | _ \\
  \text{<digit>} & = 0 | 1 | .. | 9
\end{align*}
\]

Note: an identifier is a string of one or more symbols defined in <letter> or <digit> and begins with a letter.

Example of valid identifiers: aaa990, b87b
Example of invalid identifiers: 9ab, abc9*1
5. Recursion

### Recognition algorithm

\[ \text{id}(\text{in out } w : \text{string}) : \text{boolean} \]
\[
\begin{align*}
& \quad \text{if (length of}(w) \text{ is 1)} \\
& 	\quad \text{if } w \text{ is a letter return true} \\
& 	\quad \text{else return false} \\
& \quad \text{else if (last character of } w \text{ is in } <\text{letter}> \text{ or } <\text{digit}> \\
& 	\quad \text{return } \text{id}(w \text{ minus its last character}) \\
& \quad \text{else return false}
\end{align*}
\]

Note: also, may use ADT List of chars to program the above algorithm

- Example 2: strings of the form \( A^n B^{2n} \), example: ABB, AABBBB

\(<\text{word}> = \text{empty string} | A<\text{word}>BB \)

### Recognition algorithm

\[ \text{checkWord(}\text{in out } w : \text{string}) : \text{boolean} \]
\[
\begin{align*}
& \quad \text{if(length of}(w) \text{ is 0)} \\
& 	\quad \text{return true} \\
& \quad \text{else if (1}^{st} \text{ char of } w \text{ is A and Last 2 chars of } w \text{ are B)} \\
& 	\quad \text{return checkWord}(w \text{ minus 1}^{st} \text{ and last 2 chars}) \\
& \quad \text{else} \\
& 	\quad \text{return false}
\end{align*}
\]

Note: again, may use STL List of chars to program the above algorithm
Example 3: Algebraic Expressions

- Assume the language allows only binary operators (+, -, *, /) single character variables (a, b, .. z) and parenthesis ‘(’ and ‘)’

- Example: \( x = y \ast (a + z) - b \)

- Infix expressions: every binary operator appears between its operands.

Problems:

\[
\begin{align*}
a + b \ast c & : \text{assume } \ast \text{ and } / \text{ have higher precedence over } + \text{ and } - \\
a \ast b / c & : \text{assume associate from left to right} \\
a \ast (b + c) & : \text{need to use parenthesis for } b + c
\end{align*}
\]

- Prefix: binary operator appears before its operands

Example: \( \ast a + b c = a \ast (b + c) \) \\
\( \ast + a b c = (a + b) \ast c \)

- Postfix: binary operator appears after its operands

Example: \( a b c + \ast = a \ast (b + c) \) \\
\( a b + c \ast = (a + b) \ast c \)

- Convert Infix form \(\rightarrow\) Prefix form or Postfix form

Fully parenthesized infix expression. Example (a \ast (b + c))

To prefix form: move each operator to its corresponding open parenthesis; remove all parenthesis. Example: \((a+(bc)) \rightarrow \ast a + b c\)

To postfix form: move each operator to its corresponding closed parenthesis; remove all parenthesis. Example: \((a(bc)+) \rightarrow abc++\)

Will cover better algorithms later ....
• Advantages of prefix and postfix notations:

Do not need precedence rules
Do not need association rules
Do not need to use parenthesis

• Grammar and property for prefix expressions

Grammar for prefix:

\[
<\text{prefix}> = <\text{identifier}> \mid <\text{operator}><\text{prefix}><\text{prefix}>
\]
\[
<\text{operator}> = + \mid - \mid \times \mid /
\]
\[
<\text{identifier}> = a \mid b \mid \ldots \mid z
\]

Let a string \(S[\text{first}..\text{last}]\) be a prefix expression

**Case 1**: if length of \(S = 1\) \(\Rightarrow <\text{identifier}>\)

**Case 2**: else \(S[\text{first}]\) is \(<\text{operator}>\) and \(S[\text{first}+1..\text{last}]\) consists of \(<\text{prefix}><\text{prefix}>\)

Key property:

Let \(T[1..n]\) be a string, if \(E = T[1..k]\) is a prefix expression and let \(Y = T[k+1..m]\) and \(k < m \leq n\), then \(EY\) cannot be a prefix expression, i.e. only one end point is possible for prefix \(E\).

Note: will prove this later …

So, if \(S\) is a prefix expression then \(S[\text{first}+1..\text{end}1]\) and \(S[\text{end}1+1..\text{last}]\) must be prefix expressions.
• Recognition algorithm: assume S is a string with no blank chars.

• endPre() returns the index k of the prefix expression S[first..k].

```java
endPre(in first : integer)
{
    last = S.length – 1

    if  (first < 0 or first > last)  return –1;  // failed
    else if (S[first] is an identifier )  return first  // case 1
    else if (S[first] is an operator )     // case 2
        // find the end of 1st prefix expression
        firstEnd = endPre(first + 1);

        if  (firstEnd > -1)
            // find the end of 2nd prefix expression
            return(endPre(firstEnd+1));
        else
            return -1
    else
        return –1;  // failed
}
```

• isPre() uses endPre() to determine whether S is a prefix expression

```java
isPre():Boolean  // return >=0 (PASS) or -1 (FAIL)
{
    endPosition = endPre(0);
    return (endPosition >= 0 and endPosition == S.length – 1);
}
```
5. Recursion

• Example: \( S = + * A B - C D \)

\[
\begin{align*}
\text{Start: } \text{endPre}(0); \\
S[0] &= +; \\
\text{firstEnd} &= \text{endPre}(1) \Rightarrow \\
S[1] &= *; \\
\text{firstEnd} &= \text{endPre}(2) \Rightarrow \\
S[2] &= A; \text{return } 2; \\
\text{return } (\text{endPre}(3)) \Rightarrow \\
S[3] &= B; \text{return } 3; \\
\text{// here firstEnd is } 3 \\
\text{return } (\text{endPre}(4)) \Rightarrow \\
S[4] &= -; \\
\text{firstEnd} &= \text{endPre}(5) \Rightarrow \\
S[5] &= C; \text{return } 5; \\
\text{return } (\text{endPre}(6)) \Rightarrow \\
S[6] &= D; \text{return } 6; \\
\text{End: return } 6
\end{align*}
\]
• Algorithm to evaluate a prefix expression

// assume input S is a valid prefix expression

evaluatePrefix(inout S:string):float // note : S is inout parameter
{
    ch = S[0];
    remove 1st char from S

    if (ch is an identifier) return (value of ch)

    else // let ch be an operator op

        operand1 = evaluatePrefix(S);
        operand2 = evaluatePrefix(S);
        return (operand1 op operand2);
}

Example : S =  + * 1 3 – 4 5
          0 1 2 3 4 5 6 // position

evaluatePrefix(S)
  ch = +;
  operand1 = evaluatePrefix(S) \[13-45

  ch = *;
  operand1 = evaluatePrefix(S) \[13-45

  ch = 1;
  operand2 = evaluatePrefix(S) \[3-45

  ch = 3;
  return (1*3)

  operand2 = evaluatePrefix(S) \[-45

  \[...

  return (4 –5)

  return (3+(-1))