RECURSIVE

Objective

At the end of the class students should be able to:

- Identify problem solving characterestics using recursive.
- Trace the implementation of recursive function.
- Write recursive function in solving a problem

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Introduction

- Recursion can be used to replace loops.
- Recursively defined data structures, like lists, are very well-suited to processing by recursive procedures and functions
- A recursive procedure is mathematically more elegant than one using loops.
- Sometimes procedures that would be tricky to write using a loop are straightforward using recursion.

Introduction

- Repetitive algorithm is a process wherby a sequence of operations is executed repeatedly until certain condition is achieved.
- Repetition can be implemented using loop :

while, for or do..while.

- Besides repetition using loop, C++ allow programmers to implement recursive.
- Recursive is a repetitive process in which an algorithm calls itself.

Introduction

- Recursive is a powerful problem solving approach, since problem solving can be expressed in an easier and neat approach.

- Drawback : Execution running time for recursive function is not efficient compared to loop, since every time a recursive function calls itself, it requires multiple memory to store the internal address of the function.

Recursive solution

- Not all problem can be solved using recursive.
- Problem that can be solved using recursive is a problem that can be solved by breaking the problem into smaller instances of problem, solve & combine

Understanding recursion

Every recursive definition has 2 parts:

- BASE CASE(S): case(s) so simple that they can be solved directly

- RECURSIVE CASE(S): more complex – make use of recursion to solve *smaller* subproblems & combine into a solution to the larger problem

Rules for Designing Recursive

1. Determine the **base case** - There is one or more terminal cases whereby the problem will be solved without calling the recursive function again.
2. Determine the **general case** – recursive call by reducing the size of the problem
3. Combine the base case and general case into an algorithm

Rules for Designing Recursive

Designing Recursive Algorithm

- Recursive algorithm.

```
if (terminal case is reached)            // base case
        <solve the problem>
  else                                   // general case
        < reduce the size of the problem
        and
        call recursive function >
```

Base case and general case is combined

Classic Recursive Examples

- Multiplying numbers
- Find Factorial value.
- Fibonacci numbers

Multiply 2 numbers using Addition Method

- Multiplication of 2 numbers can be achieved by using addition method.
- Example :

To multiply 8 x 3, the result can also be achieved by adding value 8, 3 times as follows:

$$8 + 8 + 8 = 24$$

Solving Multiply problem recursively

Steps to solve Multiply() problem recursively:

- Problem size is represented by variable N. In this example, problem size is 3. Recursive function will call Multiply() repeatedly by reducing N by 1 for each respective call.

- Terminal case is achieved when the value of N is 1 and recursive call will stop. At this moment, the solution for the terminal case will be compted and the result is returned to the called function.

- The simple solution for this example is represented by variable M. In this example, the value of M is 8.

Implementation of recursive function: Multiply()

```
int Multiply (int M,int N)
{
if (N==1)
return M;   else
return M + Multiply(M,N-1);
}//end Multiply()
```

Recursive algorithm

3 important factors for recursive implementation:

- There's a condition where the function will stop calling itself. (if this condition is not fulfilled, infinite loop will occur)
- Each recursive function call, must return to the called function.
- Variable used as condition to stop the recursive call must change towards terminal case.

UTM UNIVERSITI TEKNOLOGI MALAYSIA

Tracing Recursive Implementation for Multiply().

Step 1: Get the multiplication of 2 numbers.
Problem: Multiply (8,3);

Step 2: Run Multiply() function.

Sub problem1: int Multiplyint M, int N)
Value of M =8 and N =3.
Since, N ≠ 1, Multiply() will be called and the parameter value is reduced

```
return 8 + Multiply(8,3-1)
```

Step 3: Run Multiply() function.

Sub problem2: int Multiply(int M, int N)
Value of M =8 and N =2.
Since, N ≠ 1, Multiply() will be called and the parameter value is reduced

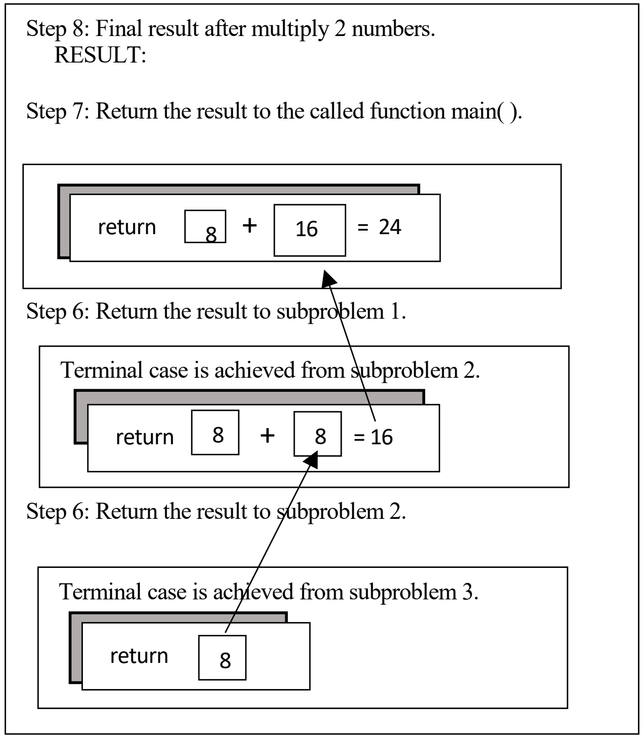
```
return 8 + Multiply(8,2-1)
```

Step 4: Run Multiply() function..

Sub problem3: : int Multiply(int M, int N)
Value of M =8 and N =1.
When N=1, terminal case is achieved.

```
return 8
```

Returning the Multiply()



Factorial Problem

- Problem : Get Factorial value for a positive integer number.
- Solution : The factorial value can be achieved as follows:

0! is equal to 1

1! is equal to 1 x *0!* = *1 x 1 = 1*

2! is equal to 2 x *1!* = *2 x 1 x 1 = 2*

3! is equal to 3 x *2!* = *3 x 2 x 1 x 1 = 6*

4! is equal to 4 x *3!* = *4 x 3 x 2 x 1 x 1 = 24*

N! is equal to N x (N-1)! For every N>0

Solving Factorial Recursively

1. The simple solution for this example is represented by the factorial value equal to 1.
2. N, represent the factorial size. The recursive process will call factorial() function recursively by reducing N by 1.
3. Terminal case for factorial problem is when N equal to 0. The computed result is returned to called function.

Factorial function

Here's a function that computes the factorial of a number N without using a loop.

- It checks whether N is equal 0. If so, the function just return 1.
- Otherwise, it computes the factorial of (N – 1) and multiplies it by N.

```
int Factorial (int N )
{ /*start Factorial*/  if (N==0)
return 1;
else
return N * Factorial (N-1);
} /*end Factorial
```

Execution of Factorial(3)

STEP 1: Get factorial 3.

Problem: `Factorial(3);`

STEP 2: Run `Factorial()`.

Sub problem 1: `int Factorial (int N)`
Value for `N=3`.
Since N ≠ 0, `Factorial()` is called by reducing the parameter value.

```
return N * Factorial(3-1);
```

STEP 3: Run `Factorial()`.

Sub problem 2: `int Factorial (int N)`
Value for `N =2`.
Since N ≠ 0, `Factorial()` is called by reducing the parameter value.

```
return N * Factorial(2-1);
```

STEP 4: Run `Factorial()`..

Sub problem 3: `int Factorial (int N)`
Value for `N =1`.
Since N ≠ 0, `Factorial()` is called by reducing the parameter value.

```
return N * Factorial(1-1);
```

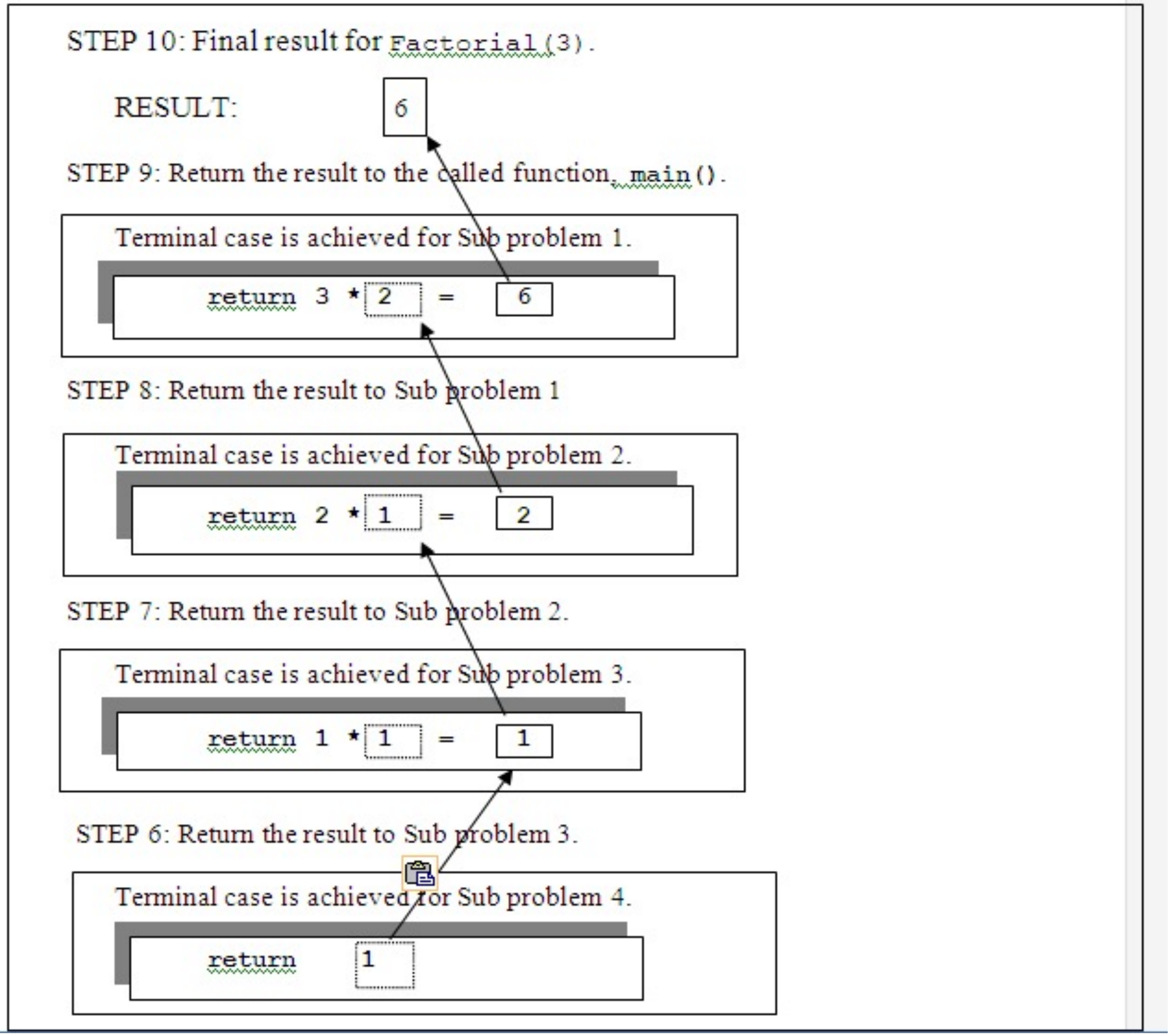
Terminal case for Factorial (3)

STEP 5: Run `Factorial()`..

Sub problem 4: `int Factorial (int N)`
Value for `N =1`.
Since `N =` 0, terminal case is achieved.

```
return  1
```

Return value for Factorial(3)



Fibonacci Problem

- **Problem : Get Fibonacci series for an integer positive.**
- **Fibonacci Siries : 0, 1, 1, 2, 3, 5, 8, 13, 21,.....**
- Starting from 0 and and have features that every Fibonacci series is the result of adding 2 previous Fibonacci numbers.
- Solution: Fibonacci value of a number can be computed as follows:

Fibonacci (0) = 0
Fibonacci (1) = 1
Fibonacci (2) = 1
Fibonacci (3) = 2
Fibonacci (N) = Fibonacci (N-1) + Fibonacci (N-2)

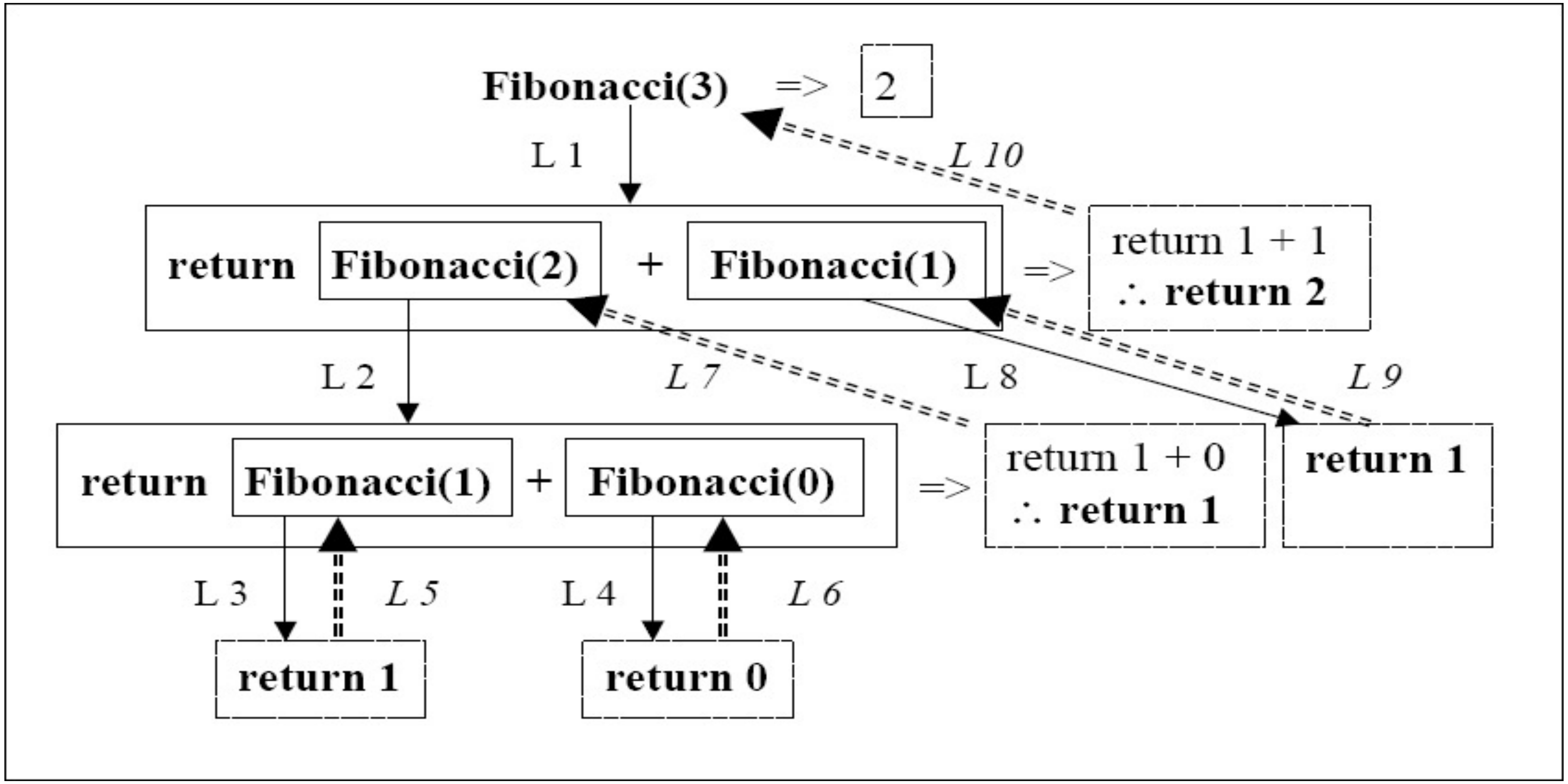
Solving Fibonacci Recursively

1. The simple solution for this example is represented by the Fibonacci value equal to 1.
2. N, represent the series in the Fibonacci number. The recursive process will integrate the call of two Fibonacci () function.
3. Terminal case for Fibonacci problem is when N equal to 0 or N equal to 1. The computed result is returned to the called function.

Fibonacci Function

```
int Fibonacci (int N )
{ /* start Fibonacci*/  if (N<=0)
return 0;  else if (N==1)
return 1;
else
return Fibonacci(N-1) + Fibonacci (N-2);
}
```

Recursive solution for Fibonacci



Infinite Recursive

- Recursion that cannot stop is called infinite recursion
- Infinite recursion occur when the recursive function:
  - Does not has at least 1 base case (to terminate the recursive sequence)
  - Size of recursive case is not changed
  - During recursive call, size of recursive case does not get *closer to a base case*

Infinite Recursive : Example

```
#include <stdio.h>
#include <conio.h>
void printIntegesr(int n);
main()
{       int number;
        cout<<"\nEnter an integer value :";
        cin >> number;
        printIntegers(number);
}
void printIntegers (int nom)
{       cout << "\Value : " << nom;
        printIntegers (nom);
}
```

1. No condition satatement to stop the recursive call.
2. Terminal case variable does not change.

Improved Recursive function

```
#include <stdio.h>
#include <conio.h>

void printIntegers(int n);  main()
{ int number;
cout<<"\nEnter an integer value :";
cin >> number;  printIntegers(number);
}
void printIntegers (int nom)


{     if (nom >= 1)
      {    cout << "\Value : " << nom;
           printIntegers (nom-2);
      }
}
```

Exercise: Give the output if the value entered is 10 or 7.

condition satatement to stop the recursive call and the cahnges in the terminal case variable are provided.

Solve this problem

Euclid's algorithm to find the Greatest Common Divisor (GCD) of a and b (a ≥ b)

- if a % b == 0, the GCD(a, b) = b
- otherwise, GCD(a, b) = GCD(b, a % b)
- Problem : Find GCD(3,8)

```
int GCD(int a, int b)
{
if (a % b == 0) { // BASE CASE
return b;
}
else
{ // recursive call  return GCD(b, a% b);
}
}
```

Solve this problem

Give the output of the recursive program below. Trace the program and show all steps involve while implementing the recursive call.

```
#include <iostream.h>  int Calc (int n)
{
if (n < 0)   return n;
else
return Calc(n-1)* Calc(n-2);
}
int main()
{   cout << Calc(5) << endl;  return 0;
}
```

Conclusion and Summary

- Recursive is a repetitive process in which an algorithm calls itself.
- Problem that can be solved using recursive is a problem that can be solved by breaking the problem into smaller instances of problem, solve & combine
- Every recursive definition has 2 parts:
  - BASE CASE: case that can be solved directly
  - RECURSIVE CASE: use recursion to solve *smaller* subproblems & combine into a solution to the larger problem

References

1. Nor Bahiah et al. *Struktur data & algoritma menggunakan C++. Penerbit UTM, 2005*
2. Richrd F. Gilberg and Behrouz A. Forouzan, "*Data Structures A Pseudocode Approach With C++*", Brooks/Cole Thomson Learning, 2001.

Thank You