SCJ2013 Data Structure & Algorithms

Recursive

Nor Bahiah Hj Ahmad & Dayang Norhayati A. Jawawi

Objectives

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

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 to replace loops.
- Not all programming language allow recursive implement, e.g. Basic language.

Introduction

- Recursive is a repetitive process in which an algorithm calls itself.
- Recursively defined data structures (like lists) are very well-suited to be processed using recursive procedure.
- A recursive procedure is mathematically more elegant than one using loops. Sometimes procedures can become straightforward and simple using recursion as compared to loop solution procedure.

Introduction

- Advantage : 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.
- Recursive solve problem by:
  1. breaking the problem into the same smaller instances of problem,
  2. solve each smallest problem and
  3. combine back the solutions.

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 and make use of recursion to:
  - break the problem to smaller sub-problems and
  - combine into a solution to the larger problem

Rules for Designing Recursive Algorithm

1. Determine the base case – is terminal case, there is one or more terminal cases whereby the problem will be solved and stop to call recursive function.
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

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 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:

$$8 + 8 + 8 = 24$$

Implementation of `Multiply()` using loop

```
int Multiply(int M,int N)
{ for (int i=1,i<=N,i++)
     result += M;
  return result;
}//end Multiply()
```

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 computed 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.

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 Multiply(int 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..

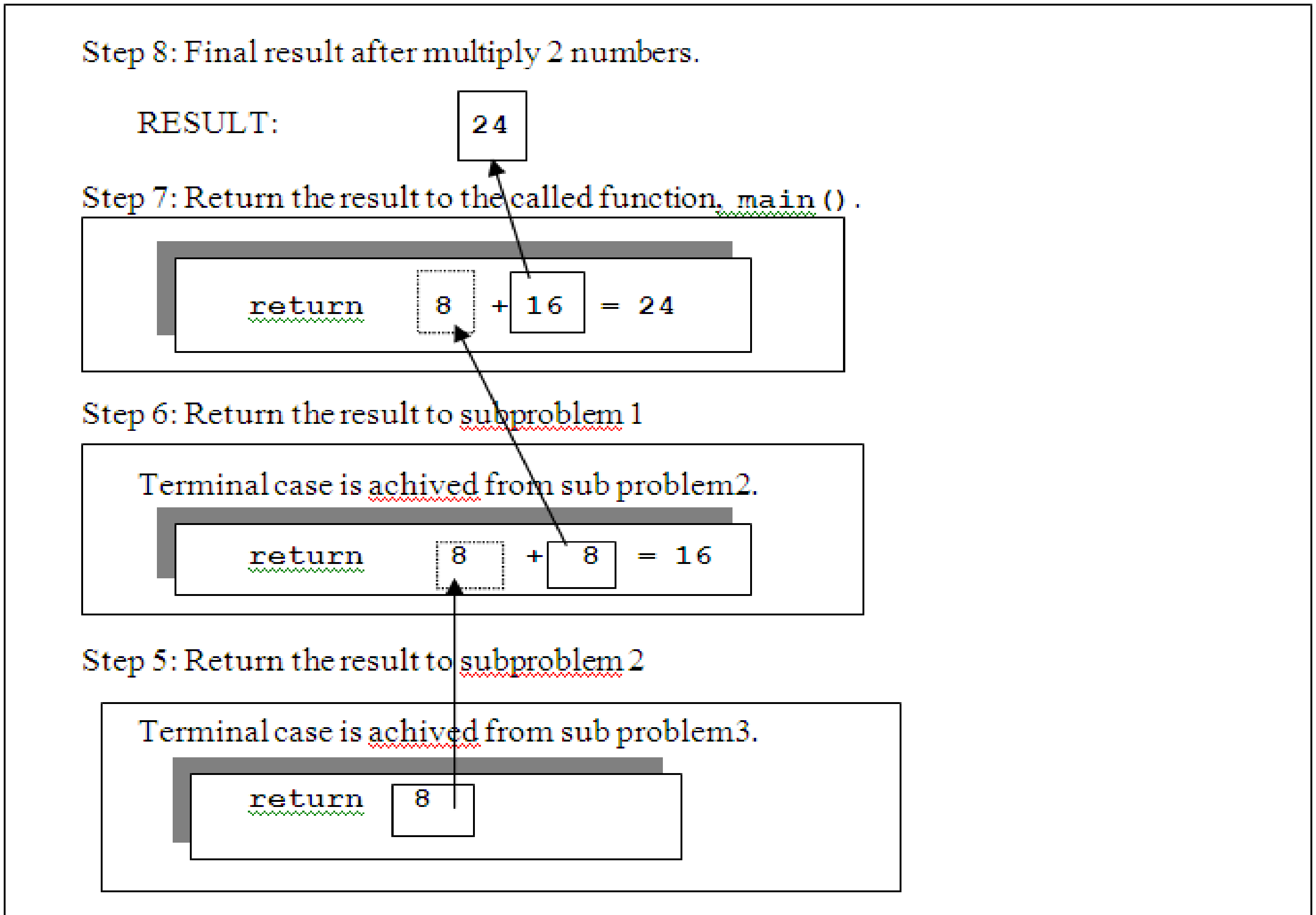
Sub problem3: : `int Multiply(int M, int N)`

Value of **M** =8 and **N** =1.

When N=1, terminal case is achieved.

```
return 8
```

Returning the `Multiply()` result to the called function



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

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

- 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.

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)`

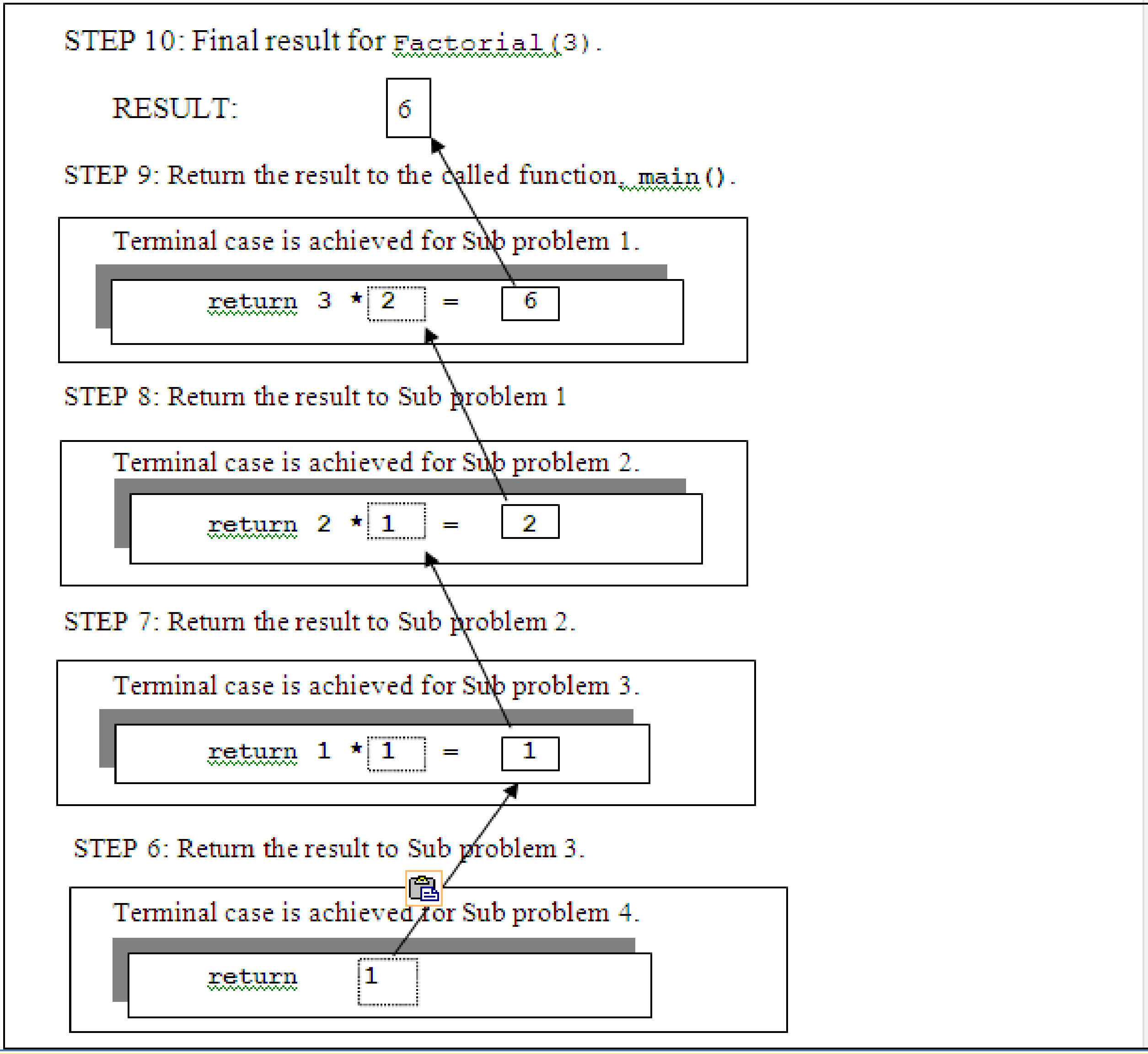
Value for `N` =1.

Since `N` = 0, terminal case is achieved.

```
return  1
```

Execution of `Factorial(3)`

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,…..
- Start from 0 and 1
- 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

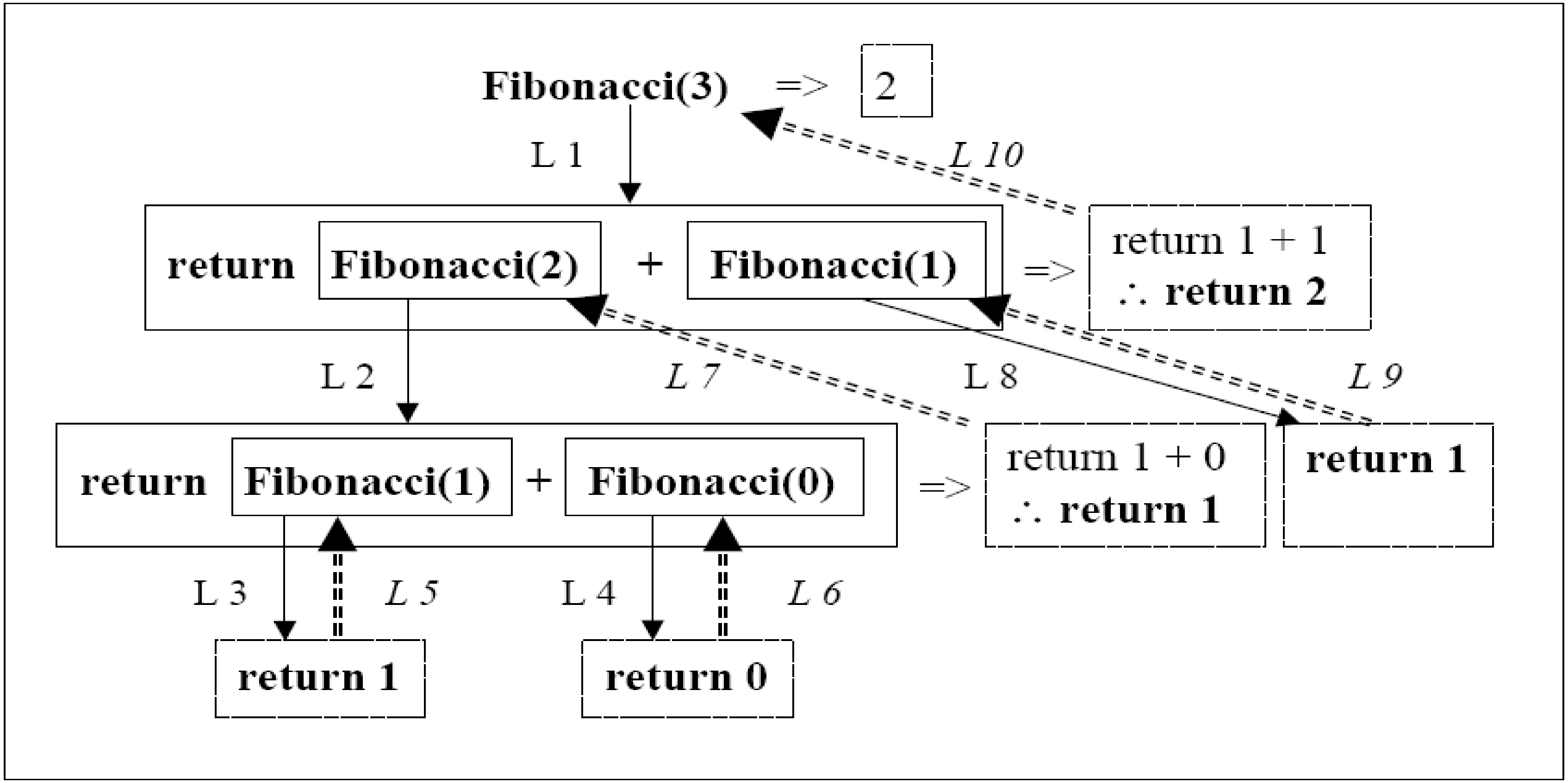
1. The simple solution for this example is represented by the Fibonacci value equal to 1.
2. N, represent the series in the Fibonacci number. The recursive process will integrate the call of two **`Fibonacci ()`** function.
3. Terminal case for Fibonacci problem is when N equal to 0 or N equal to 1.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

Implementation of `Fibonacci()`

- Passing and returning value from function.



Infinite Recursive

- Impossible termination condition

- How to avoid infinite recursion:
  - must have at least 1 base case (to terminate the recursive sequence)
  - each recursive call must 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 changes in the terminal case variable are provided.

Conclusion and Summary

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