جامعة الإسراء
ISRA UNIVERSITY

Unit 3
Recursion

Student will (Unit ILOs):

1. Learn the fundamentals of Recursion
2. Learn how to write recursive algorithms
3. Learn the difference between Recursion and Looping

Introduction

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

- Recursion can be used to replace loops.
- Recursively defined data structures, like lists, are very well-suited for processing by recursive 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

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

Designing Recursive Algorithm

```
ReturnValueType     Method'sName   (InputParameters)
{
   ReturnValueType     Result
    if (terminal case is reached)                          // Termination case
            Then Result= Termination Value
      else                                                 // recursive case
            Result = Method'sName (New Arrguments) + ….
      endif
      return  Result
 }
```

3 important factors for recursive implementation

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

A Recursive Method is a method that it calls itself

Example 1: Factorial

int x = factorial (4);

F	n
1	4
4	3
12	2
24	1

x = 24

Non – Recursive Factorial

```
int factorial ( int n )
{
    int F = 1;
    while ( n > 1 ) {
        F = F * n;
        n = n – 1;
    }
    return F;
}
```

A Recursive Method is a method that it calls itself

Example 1: Factorial

$0! = 1$
$1! = 1$ — Termination Cases

$n! = n * (n-1)!$ — Recursive Case

$n! = n * (n-1)!$

$(n-1)! = (n-1) * (n-2)!$

$(n-2)! = (n-2) * (n-3)!$

...

$\boxed{\text{Hit the base !}} \rightarrow (1)! = 1$

Recursive Factorial

```
int factorial ( int n )
{
  if ( n = =  1 ) {
    return n;
  }
  else {
    return n* factorial ( n-1 );
  }
}
```

int x = factorial (4);

	First Copy
n = 4	int factorial (int n)
	{
	if (n = = 1) {
	return n;
	}
	else {
4 * ?	return n* factorial (n-1);
	}
	}

int x = factorial (3);

n = 4

First Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

4 * ?

n = 3

Second Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

3 * ?

int x = factorial (2);

n = 4

First Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

4 * ?

n = 3

Second Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

3 * ?

n = 2

Third Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

2 * ?

int x = factorial (1);

n = 4

4 * ?

First Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n; }
5	else {
6	return n* factorial (n-1);
7	}
8	}

n = 3

3 * ?

Second Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n; }
5	else {
6	return n* factorial (n-1);
7	}
8	}

n = 2

2 * ?

Third Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n; }
5	else {
6	return n* factorial (n-1);
7	}
8	}

n = 1

1

Fourth Copy	
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n; }
5	else {

int x = factorial (2);

n = 4

4 * ?

	First Copy
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

n = 3

3 * ?

	Second Copy
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

n = 2

2 * 1

	Third Copy
1	int factorial (int n)
2	{
3	if (n = = 1) {
4	return n;
5	}
6	else {
7	return n* factorial (n-1);
8	}
9	}

int x = factorial (3);

	First Copy	
n = 4	1	int factorial (int n)
	2	{
	3	if (n = = 1) {
	4	return n;
	5	}
	6	else {
4 * ?	7	return n* factorial (n-1);
	8	}
	9	}

	Second Copy	
n = 3	1	int factorial (int n)
	2	{
	3	if (n = = 1) {
	4	return n;
	5	}
	6	else {
3 * 2	7	return n* factorial (n-1);
	8	}
	9	}

int x = factorial (4);

	First Copy	
n = 4	1	int factorial (int n)
	2	{
	3	if (n = = 1) {
	4	return n;
	5	}
	6	else {
4 * 6	7	return n* factorial (n-1);
	8	}
	9	}

Example 2: Multiply two numbers using Addition Operation

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

- Problem size is represented by variable N. In this example, problem size is 3.
- Termination case is achieved when the value of N is 1, other value make the recursive case.

By Using loop

```
int Multiply (int M, int N)
{

   for (i=1; 1<=N; i++)
   result+=M


   return result

}
```

Recursively:

```
int Multiply (int M,int N)
{
   int Result
   if (N==1)
     then Result=M;
   else
     Result=M + Multiply(M,N-1);
   endif
   return  Result
}
```

Tracing the

Recursive

Implementation

or 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.

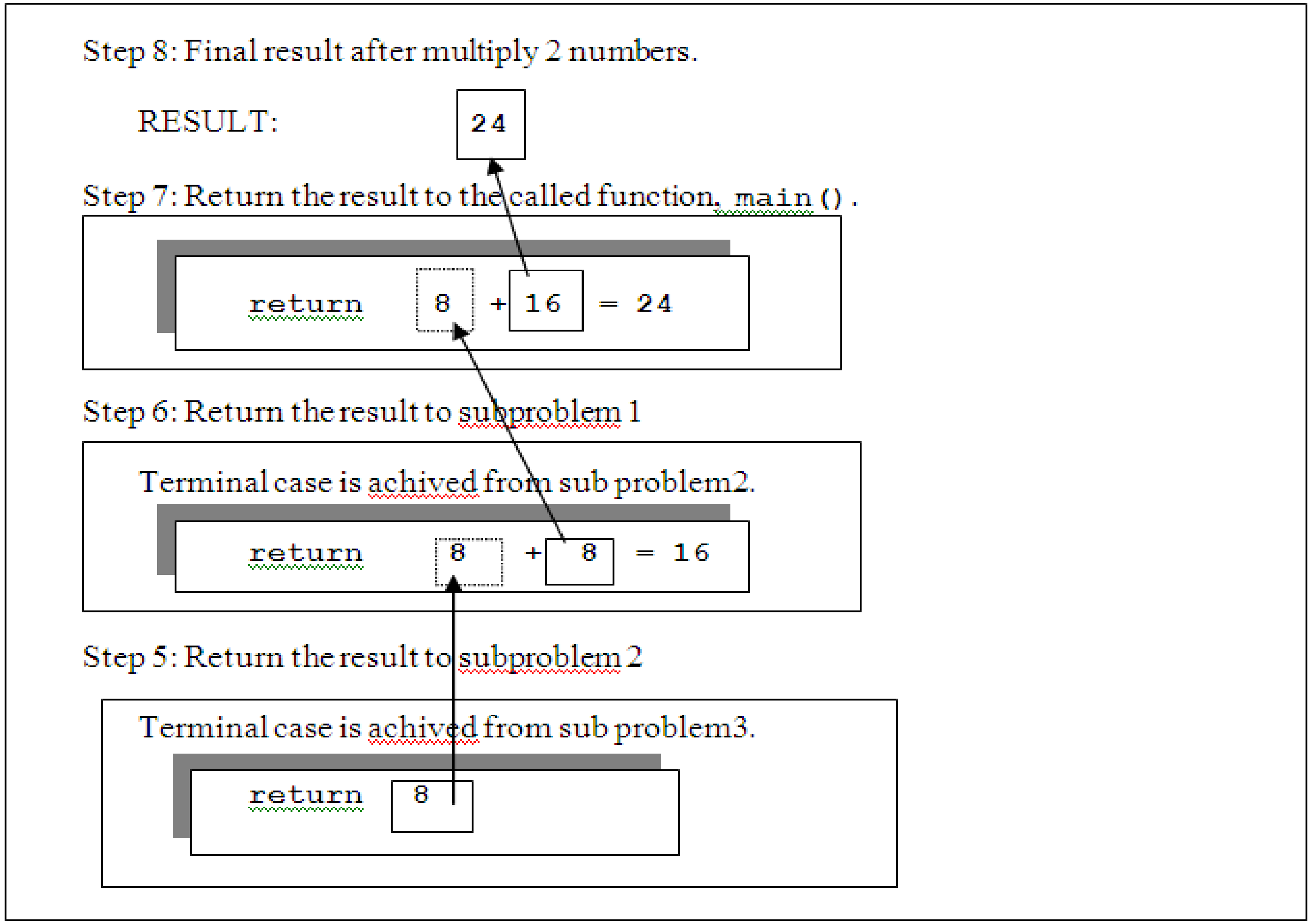
```
return 8
```

Tracing the

Recursive

Implementation

or Multiply.



Example 3: Sum of Squares

The sum of squares is the sum of all squared numbers starting from the value (n^2) up to (m^2), where $n < m$.

$$SumS = n^2 + (n+1)^2 + (n+2)^2 + . . . + m^2$$

Example 3: Sum of Squares

int x = SumS (2, 4);

sum	n
0	2
4	3
13	4
2	5
9	

x = 29

Using Loop

```
int SumS ( int n, int m )
{
    int sum = 0;
    while ( n <= m ) {
        sum= sum + n * n;
        n = n + 1;
    }
    return sum;
}
```

Sum = (2*2) + (3*3) + (4*4) = 4 + 9 + 16

Example 3: Sum of Squares

$$SumS(n, m) = \underbrace{SumS(n+1,m)}_{SumS(n+2,m) + (n+1)^2} + n^2$$

$$SumS(n+2,m) + (n+1)^2$$

$$(n+2) * (n+2)$$

Recursive SumS

```
int SumS ( int n, int m )
{
   if ( n = =  m ) {
      return n * n;
   }
   else {
      return n* n + SumS ( n+1, m );
   }
}
```

int x = SumS (2, 4);

First Copy

```
int SumS ( int n, int  m )
{
   if ( n = =  m ) {
      return n * n;
   }
   else {
      return n* n + SumS ( n+1,m );
    }
}
```

n = 2, m = 4

4 + ?

int x = SumS (2, 4);

	First Copy
1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

n = 2,
m = 4

4 + ?

	Second Copy
1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

n = 3,
m = 4

9 + ?

int x = SumS (2, 4);

First Copy (n = 2, m = 4)

1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

4 + ?

Second Copy (n = 3, m = 4)

1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

9 + ?

Third Copy (n = 4, m = 4)

1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

4 * 4

int x = SumS (2, 4);

	First Copy
1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

n = 2,
m = 4

4 + ?

	Second Copy
1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

n = 3,
m = 4

9 + 16

int x = SumS (2, 4);

First Copy	
1	int SumS (int n, int m)
2	{
3	if (n = = m) {
4	return n * n;
5	}
6	else {
7	return n* n + SumS (n+1,m);
8	}
9	}

n = 2, m = 4

4 + 25

int x = SumS (2, 4);

First Copy

```
int SumS ( int n, int  m )
{
   if ( n = =  m ) {
      return n * n;
  }
   else {
      return n* n + SumS ( n+1,m );
    }
}
```

n = 2, m = 4

4 + 25

x = 29