Introduction to Recursion

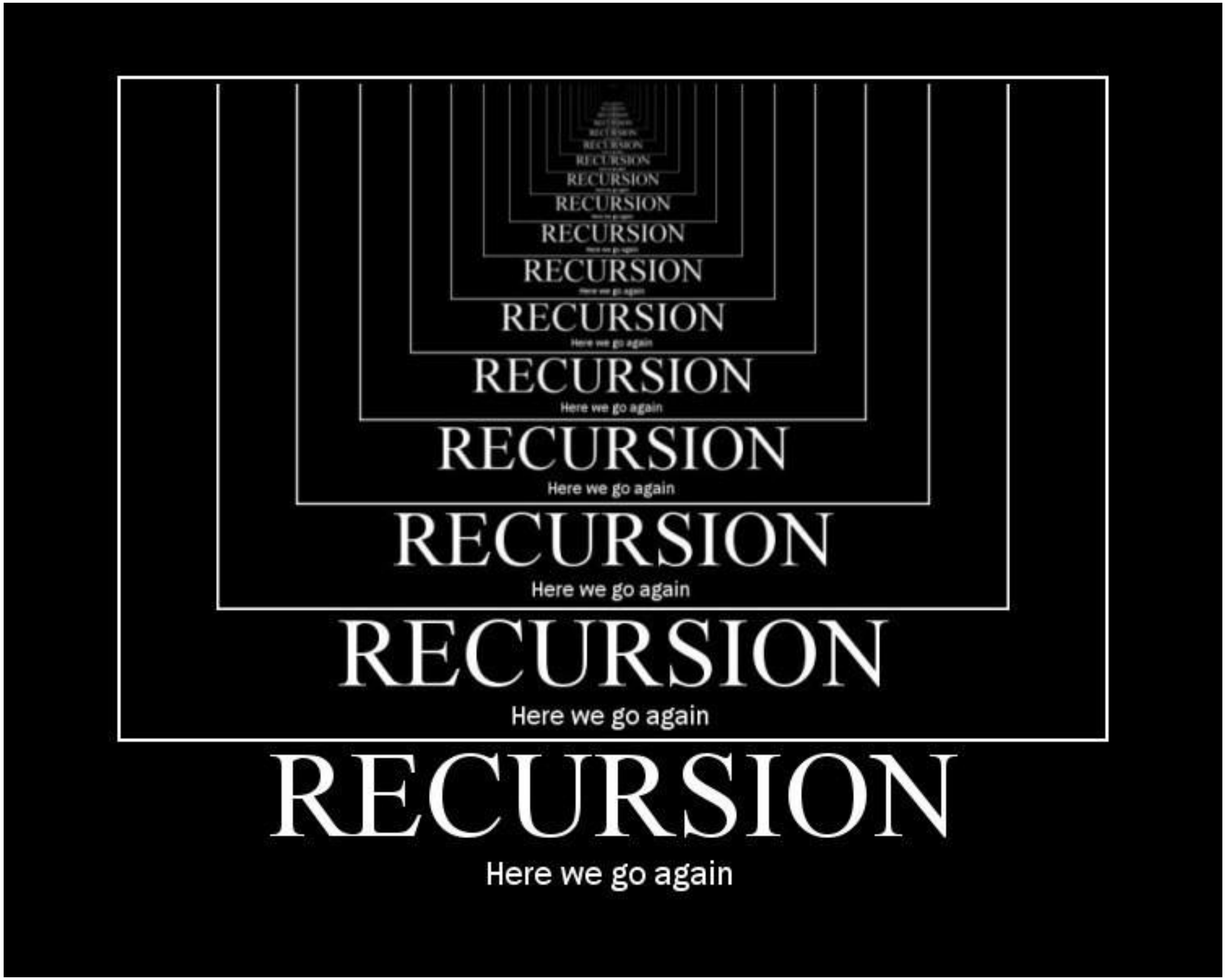
Dr. Robert Amelard
(adapted from Dr. Igor Ivkovic)

ramelard@uwaterloo.ca

Objectives

- Introduction to Recursion
- Designing Recursive Algorithms
- Bottom-Up vs. Top-Down Problem Solving

re•cur•sion [ri-kur-zhuhn]
n. See recursion.



Introduction to Recursion /1

- **Recursion as a problem solving strategy:**
  - A computing problem can be solved by representing it as smaller versions of itself

- **For instance, consider the factorial function (n!)**
  - To compute 4!, we need to compute 4 x 3 x 2 x 1
  - However, the same function can be represented as 4 x 3!
  - That is, we can represent the same problem using a smaller version of itself (3!), which is then combined (4 x) to obtain a solution to the original problem (4!)

- **This approach of representing and solving a computing problem using smaller, recurrent versions of itself is referred to as recursion**

Introduction to Recursion /2

- “Rules” of Recursion
  1. You must always have some base cases, which can be solved without recursion.
  2. For cases that are to be solved recursively, the recursive call must always be a case that makes progress toward a base case.
  3. You’ve gotta believe!

Introduction to Recursion /3

- Example (Trivial):

```
void HelloWorld(int count)
{
    if (count < 1)
        return;
    cout << "Hello World!" << endl;
    HelloWorld(count - 1);
}

int main()
{
    HelloWorld(3);
}
```

```
int factorial(int n)
{
    if (n == 0)
        return 1;

    int calculation = n * factorial(n - 1);
    return calculation;
}
```

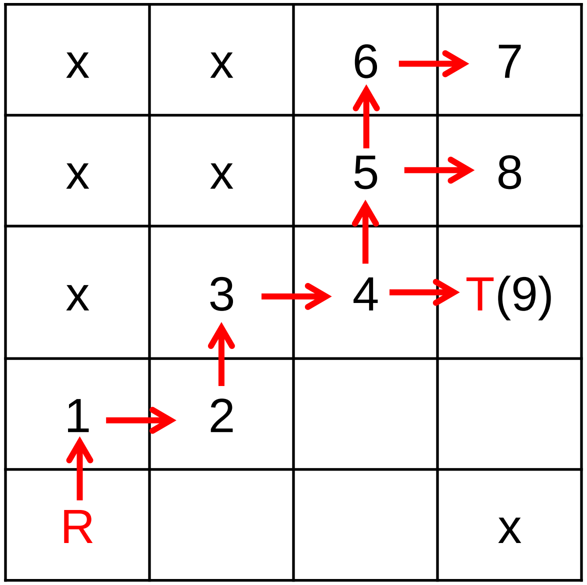
Introduction to Recursion /4

- Example (Robot path finding):

```
// A robot finding a target, that can only traverse up and right
bool findTarget(x, y)
{
    if (checkForTargetAt(x, y) == true)
    {
        reportCoordinates(x, y);
        return true;
    }
    if (isOutOfBounds(x, y) == true) return false;

    if (findTarget(moveUp(x, y))) return true;
    if (findTarget(moveRight(x, y)) return true;

    return false;
}
```



Introduction to Recursion /5

- **Recursion is a useful technique to solve a number of computing problems in relatively simple manner**
  - Theoretically, any recursive algorithm can be represented non-recursively (iteratively)
  - However, a recursive solution may be shorter and simpler to write, especially for problems that exhibit recursion

- **Discussion question:**
  - Where can we observe recursive (repeating) behaviour in real life?

Introduction to Recursion /6

- **At the core of recursion is the solution to a problem being represented by smaller versions of itself**
  - These smaller versions can then be represented by even smaller versions of their own, and so on
- **Call Trees:**
  - Allow visualization of recursion, where each version of the problem spawns smaller representations of itself
  - This is represented as trees branches branching out from a tree

Introduction to Recursion /7

- **Example: the Fibonacci series**
  - $F_n = F_{n-1} + F_{n-2}$, where n > 1 (recursive case)
  - $F_0 = 0; F_1 = 1$ (base cases)

  - To compute F_2, we use recursion as

$$F_2 = \underbrace{F_1}_{1} + \underbrace{F_0}_{0} = 1$$

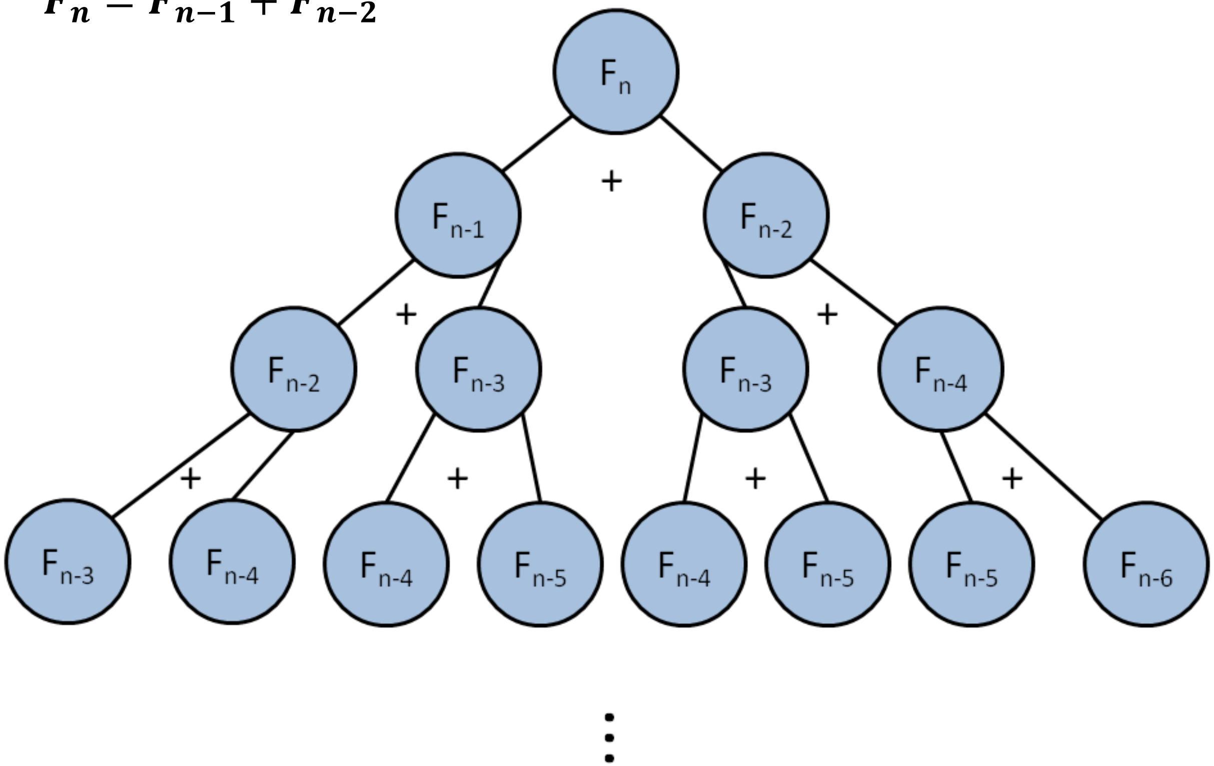
  - Notice that the original problem (deriving F_2) is represented as smaller versions of itself (F_1 and F_0)

  - Also, notice that the recursive behaviour should eventually stop, or we continue repeating it forever
  - That is, the recursion application should terminate with the base case or base cases of the problem

Introduction to Recursion /8

- **Call tree for the Fibonacci series**
  - The recursion of the Fibonacci series can be visualized with a call tree

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



Designing Recursive Algorithms /1

- **When designing recursive algorithms, answer the following two questions:**
  - Q1: How to represent the given problem using smaller versions of itself? (i.e., how to design recursive case(s))
  - Q2: When does the problem reach its ending point? (i.e., how to design base case(s))

- **To answer Q1, define what “smaller” means**
  - Smaller could mean smaller input, but more precisely, smaller means closer to the ending point of the problem

- **To answer Q2, define what is the ending point**
  - A problem reaches its ending point when there is a trivial answer to the problem, and refining it further would not be needed or would lead to undefined answers

Designing Recursive Algorithms /2

- **Once Q1 and Q2 have been answered, we can design the algorithm and represent it in code**
  - A template for writing the algorithm in code is given below

```
<return_value_of_this_call> <name_of_the_call>(<input>)
{
       if (input signifies the ending point)
              <answer_to_the_second_question>

       <answer_to_the_first_question>
}
```

  - That is, depending on the input, the corresponding program either goes into the base case (answer to Q2), or the recursive case (answer to Q1)

Designing Recursive Algorithms /3

- **Exercises:**
  - Design a recursive algorithm that computes n-th Fibonacci number
  - Design a recursive algorithm that counts the number of nodes in a list from a given node
  - Design a recursive algorithm that never terminates

Designing Recursive Algorithms /4

- **Let us approach this problem using bottom-up problem solving, using the following steps:**
  - (Step1) Solve the base cases, and encode them in code
  - (Step2) Address the general (recursive) cases, and encode those in code too
  - (Step3) Group solutions for Step1 and Step2 into a function, develop test cases to check correctness, and refine the code until it passes all the required tests

Designing Recursive Algorithms /5

- **Exercise: Design a recursive algorithm that computes n-th Fibonacci number**

```
int fibonacci(int n) {
	if (n <= 0) { // base case1
		return 0;
	} else if (n == 1) { // base case2
		return 1;
	} else if (n > 1) { // recursive case
		return fibonacci(n-1) + fibonacci(n-2);
	}
}
```

Designing Recursive Algorithms /6

- **Let us approach this problem using top-down problem solving, using the following steps:**
  - (Step1) Address the general (recursive) cases, and encode them in code
  - (Step2) Keep dividing the problem into smaller versions until base cases are reached; once the base cases are reached and solved, encode them in code
  - (Step3) Group solutions for Step1 and Step2 into a function, develop test cases to check correctness, and refine the code until it passes all the required tests

Designing Recursive Algorithms /7

- **Exercise: Design a recursive algorithm that counts the number of nodes in a list from a given node**

```
        // recursive case: count the number of nodes
        // by adding 1 to the previous node count
               return 1 + numberOfNodes(node->getNext());
```

```
        // iteration to base case: keep removing one node
        // at a time until no nodes are left; then return 0
               return 0;
```

```
int numberOfNodes(Node* node) {
       if (node) // recursive case
              return 1 + numberOfNodes(node->getNext());
       else // iterate to the base case
              return 0;
}
```

Designing Recursive Algorithms /8

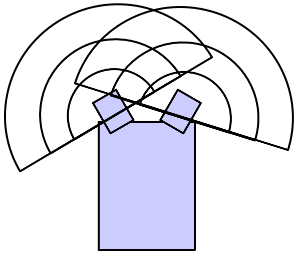
- **Exercise: Design a recursive algorithm that never terminates (infinite recursion)**

```
int badRecursion(int a, int b) {
        if (a == 0 && b == 0) // base case
                return 0;
        else // recursive case
                return badRecursion(a * 2, b - 1);
}
```

  - The above algorithm never terminates for certain inputs
  - For example, badRecursion(5, 6) never terminates, and eventually crashes the program with a stack overflow

Designing Recursive Algorithms /9

- **Challenge Question:**
  - A robot has an on-board radar sensor that can sense whether the object of interest is to the left or to the right, but cannot sense how far away it is. Design a recursive algorithm for finding the target object. Assume you have the following methods:

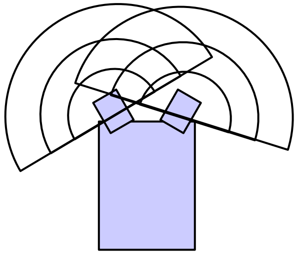


```
class RadarBot
{
public:
    ...
    bool pingLeft();  // returns true if object is to left of robot
    bool pingRight();  // returns true if object is to left of robot
    bool isObjectHere();  // looks for object at current location

    void moveLeft();
    void moveRight();

private:
    ...
};
```

Designing Recursive Algorithms /10



```
bool RadarBot::findTarget(int x, int y)
{
    // base case




    // general case










}
```

Food for Thought

- **Read:**
  - Chapter 4 (Recursion) from the course handbook

- **Additional Readings:**
  - Chapter 9 (Recursive Thinking) from “Data Structures and Other Objects Using C++” by Main and Savitch