RECURSIVE FUNCTIONS

Recursive Functions

- A recursive function is a function that calls itself.
- Recursion may provide a simpler implementation than a function that iterates (loops) to calculate an answer
  - By calling itself (and the new copy calling ***itself***), multiple iterations are automatically created and handled by the computer hardware's function-call-stack mechanism

- For example, to print a text triangle:

```
[]
[][]
[][][]
[][][][]
```

Recursive Function Example

- We use a function defined as:

```
void print_triangle(int side_length)
```

- for this output:

```
[]
[][]
[][][]
[][][][]
```

The function call will be: `print_triangle(4);`

- This is the pseudocode of a recursive version, for an arbitrary side length:

If side length < 1, return.

Else, call print_triangle with side length = side length -1.

Then print a line consisting of side length [] symbols.

Recursive Function C++ Code

```
void print_triangle(int side_length)
{
   if (side_length < 1) { return; }
   print_triangle(side_length - 1);
   for (int i = 0; i < side_length; i++)
   {
      cout << "[]";
   }
   cout << endl;
}
```

A recursive function works by calling itself with successively simpler input values.

Recursive Function Rules

Two requirements ensure that the recursion is successful:

1. Every recursive call must simplify the task in some way.
2. There must be special cases to handle the simplest tasks directly.

The `print_triangle()` function calls itself again with smaller and smaller side lengths. Eventually the side length must reach `0`, and the function stops calling itself.

Tracing the Calls to the Recursive Function

The call print_triangle(4) calls print_triangle(3).

The call print_triangle(3) calls print_triangle(2).

The call print_triangle(2) calls print_triangle(1).

The call print_triangle(1) calls print_triangle(0).

The call print_triangle(0) returns, doing nothing.

The call print_triangle(1) prints [].

The call print_triangle(2) prints [][].

The call print_triangle(3) prints [][][].

The call print_triangle(4) prints [][][][].

How to Think Recursively

Pretend that “someone else” will do most of the heavy lifting.

Just focus on reducing the problem to a simpler one with a call to the same function with smaller inputs.

You only need to figure out the last step: how to include this solution with simpler inputs into a solution for the whole problem.

Then include the exit case with no additional call, when the input reaches the limit, so that the recursion eventually stops.

Example: Computing the Sum of the Digits of a Number:

Design a function `int digit_sum(int x)` that computes the sum of the digits of an integer n.

`digit_sum(1729)` would equal 1 + 7 + 2 + 9 = 19

How to Think Recursively

Step 1 Look for simpler input that can be solved by the same task, and whose solution is related to the original task.

The key to finding a recursive solution is reducing the input to a simpler input for the same problem.

Break the input into parts that can themselves be inputs to the problem:

-- Save the last digit with: 1729 `% 10 = 9`

-- Remove the last digit and re-call with the remaining digits as input: 1729 `/ 10 = 172`

The digit sum of 172 is directly related to the digit sum of 1729.

How to Think Recursively

Step 2 Combine solutions with simpler inputs into a solution of the original problem.

When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

In your mind, consider the solutions for the simpler inputs that you have discovered in Step 1. Don't worry how those solutions are obtained. Simply have faith that the solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

How to Think Recursively

Step 2 continued:

In the case of the digit sum task, ask yourself how you can obtain `digit_sum(1729)` if you know `digit_sum(172)`.

You simply add the last digit (`9`), and you are done. How do you get the last digit? As the remainder `n % 10`. The value `digit_sum(n)` can therefore be obtained as:

```
digit_sum(n / 10) + n % 10
```

Don't worry how `digit_sum(n / 10)` is computed. The input is smaller, and therefore it just works.

Total that saved digit plus the return from the call.

Return the total.

How to Think Recursively

Step 3 Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Find solutions to the simplest inputs (the stopping points). That is usually very easy.

Look at the simplest inputs for the **`digit_sum`** test:

a number with a **`single digit`** **OR** a **`zero`**

A number with a single digit is its own digit sum, so you can stop the recursion when **`n < 10`**, and return **`n`** in that case.

And you can simply terminate the recursion when **`n`** is **`zero`**.

How to Think Recursively

Step 4 Implement the solution by combining the simple cases and the reduction step.

Now you are ready to implement the solution. Make separate cases for the simple inputs that you considered in Step 3. If the input isn't one of the simplest cases, then implement the logic you discovered in Step 2:

```
int digit_sum(int n)   //Complete function
{
   // Special case for terminating
   // the recursion:
   if (n == 0) { return 0; }
   // General case:
   return digit_sum(n / 10) + n % 10;
}
```

How to Think Recursively: the Code and a Trace

```
int digit_sum(int n)
{
    // Special case for terminating the recursion
    if (n == 0) { return 0; }
    // General case
    return digit_sum(n / 10) + n % 10;
}
```

- The call `digit_sum(1729)` calls `digit_sum(172)`.
  - The call `digit_sum(172)` calls `digit_sum(17)`.
    - The call `digit_sum(17)` calls `digit_sum(1)`.
      - The call `digit_sum(1)` calls `digit_sum(0)`.
        - » The call `digit_sum(0)` returns 0.
      - The call `digit_sum(1)` returns 1.
    - The call `digit_sum(17)` returns 1+7 = 8
  - The call `digit_sum(172)` returns 8+2 = 10
- The call `digit_sum(1729)` resumes and returns 10+9 = 19.

SUMMARY

Understand recursive function calls and implement recursive functions:

- A recursive computation solves a problem by using the solution of the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest inputs.
- The key to finding a recursive solution is reducing the input to a simpler input for the same problem.
- When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.