Recursive FUNCTIONS (RF)

"In order to understand recursion, you must first understand recursion. Crazy, isn't it ?"

https://arnoldzwicky.org/category/syntax/recursion/

Recursive Algorithm

- A recursive solution describes a procedure for a particular task in terms of applying the same procedure to a similar but smaller task.
- Must have a base case, which is when the task is so simple that no recursion is needed. This is also called *termination* case.
- Recursive calls must eventually converge to a base case.

REMARKS: Functions with a recursive algorithms must have the following:

1. Base Case (i.e., when to stop): one or more base cases
2. Work toward Base Case
3. Recursive Call (i.e., function calls itself)

Recursive Functions: Example

Review algorithm to compute factorials (no function, no recursion). Review a construction of an standard function

Properties of Factorials:

- N! = N (N-1)!

  N!=N(N-1)(N-2)!

  N!=N(N-1)(N-2)….0!

- If N=0, then 0!=1

The underlying principle is very simple: a function that can call itself:

```
function f = iFactorial(N)
% Computes the factorial of N
        if (N==0)
                f= 1;
        else
                f= N*iFactorial(N-1);
        end
end
```

Base case

Calls itself with smaller value of the parameter

Recursive Functions: Example

Two EQUIVALENT versions:

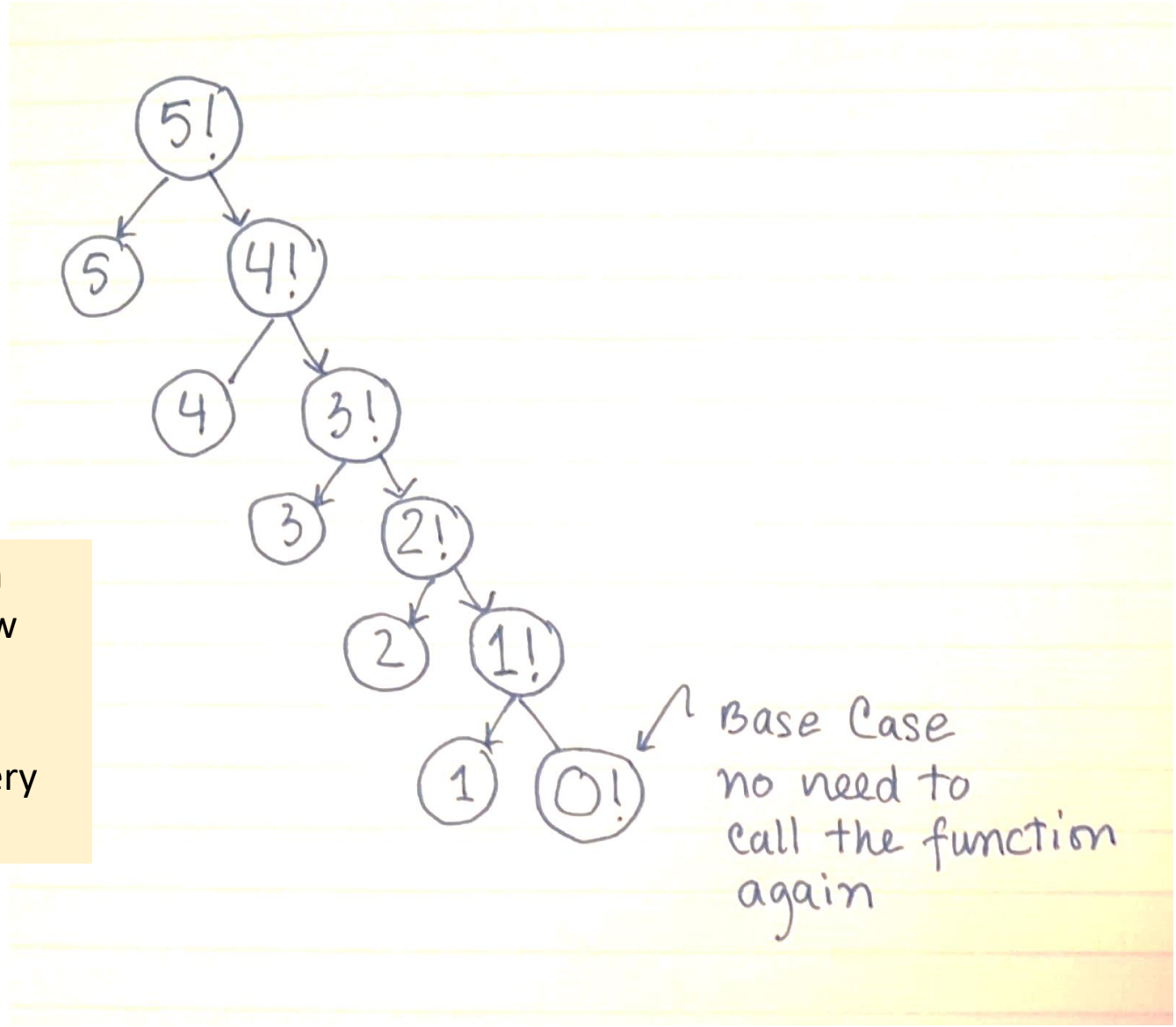
```
function f = iFactorial(N)
% Computes the factorial of N

if (N==0)
   f= 1;
else
   f = N * iFactorial(N-1);% function call
end
end
```

```
function f = iFactorial(N)
% Computes the factorial of N

  if N>0
      f = N*iFactorial(N-1); % function call
  else
      f = 1;
  end
end
```

How does it works?



We can use the structure of an expression **TREE** to explain how the recursion works.

Standard flowcharts are not very good at explaining recursion

Recursive Trace:

This graphic is to show how the computation progresses holding the execution in each call until the base case.

```
function  f  =  iFactorial  (N)
  if N>0
        f= 5 * iFactorial(4);
  else
     f=1;
  end
end
```

5 → 4 → 3 → 2 → 1 → 0

f=4*iFactorial(3);

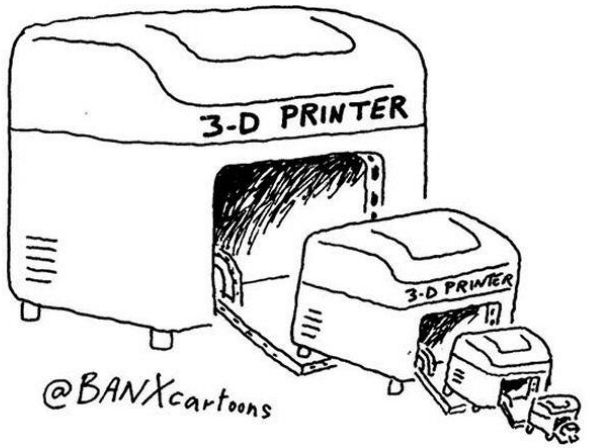
f=3*iFactorial(2);

f=2*iFactorial(1);

f=1*iFactorial(0);

f=1;

resume:
f=5*4*3*2*1*1



Recursion Limit (1)

- Design your algorithm so that it always find a final solution and stops calling itself or you will go infinite loop.
- Matlab is ready for the ‘infinite’ scenario and you will get an error before your computer goes crazy, like so :

Out of memory. The likely cause is an infinite recursion within the program.

Error in myrecursivefun (line 7)

```
retVal = myrecursivefun(inVal, recursions);
```

Recursion Limit (2)

```
% CrashMatlab.m
clc, clear
myrecursivefun(1, 70000)
```

OUTPUT

Out of memory. The likely cause is an infinite recursion within the program.

Error in myrecursivefun (line 7)

retVal = myrecursivefun(inVal, recursions);

```
function retVal = myrecursivefun(inVal, recursions)
% A recursive function to experiment with
% the recursion limit in matlab
    recursions = recursions - 1;
    inVal = inVal + 1;
    if recursions > 0
        retVal = myrecursivefun(inVal, recursions);
    else
        retVal = inVal;
    end
end
```

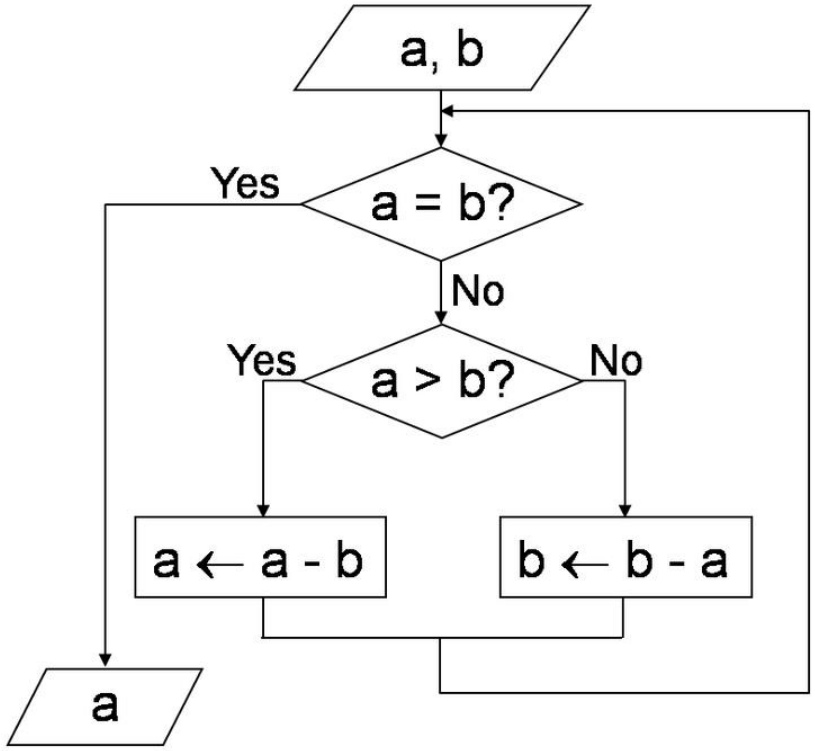
FIXED

```
% CrashMatlab.m
clc, clear
set(0,'RecursionLimit', 65000);
myrecursivefun(1, 65000)
```

However the “fixed” code will be very slow as you are altering the system default; DON’T TRY

Euclid's Algorithm: Another Example

- Finds the greatest common factor (GCF) of two positive integers
- Circa 300 BC



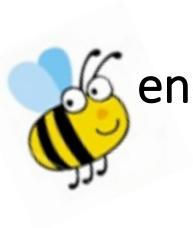
Euclides gcd

With iterations:

```
function [R] = gcd(a,b)
   % Computes gcd by Euclidean method

   while a~=b
      if a>b
         a=a-b;
      elseif a<b
         b=b-a;
      end
   end
   R=a;

end
```



With Recursion:

```
function [R] = gcdR(a,b)
   % Recursive gcd function By Euclidean method

   if a==b
      R=a;
   elseif a>b
      R=gcdR(a-b,b);
   elseif a<b
      R=gcdR(b,b-a);
   end

end
```

The recursive tale:

A child couldn't sleep, so her mother told a story about a little frog,

who couldn't sleep, so the frog's mother told a story about a little bear,

who couldn't sleep, so bear's mother told a story about a little weasel

...who fell asleep.

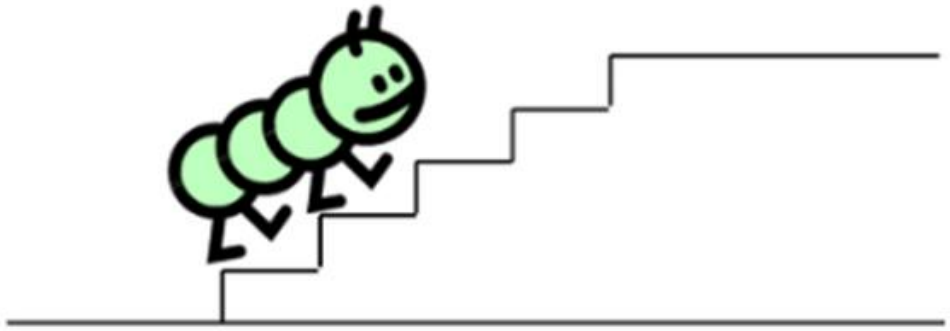
...and the little bear fell asleep;

...and the little frog fell asleep;

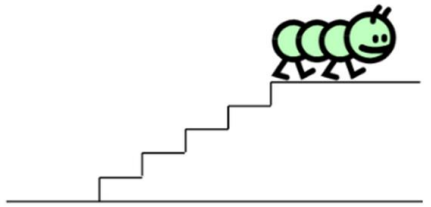
...and the child fell asleep.

Recursive case: more steps to climb ⇒

1. Step up one step
2. **Climb steps**



Base case: if no steps to climb ⇒ stop



Procedure: **Climb steps**

Pros and Cons

The advantages of recursive functions are:

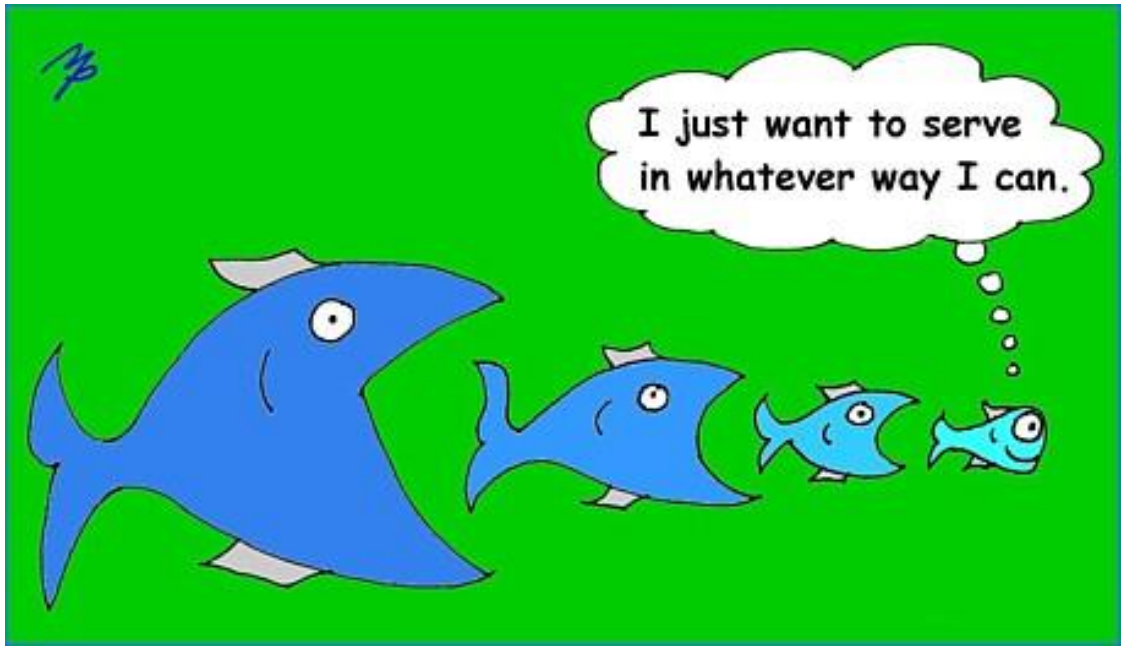
- Avoidance of unnecessary calling of functions.
- A substitute for iteration where the iterative solution is very complex. For example to reduce the code size for Tower of Honai application, a recursive function is best suited.
- Extremely useful when applying the same solution

The disadvantages of Recursive functions:

- A recursive function is often confusing.
- The exit point must be explicitly coded.
- It is difficult to trace the logic of the function.

Exercises

- Exercises with solutions in document:
- “Recursive Functions.doc”



Some References

- https://medium.freecodecamp.org/recursion-demystified-99a2105cb871

How Does it Work

$$4^2 + 3^2 + 2^2 + 1^2 + 0^2$$

- Implemented on a computer as a form of iterations, but hidden from the programmer
- Assisted by the system stack

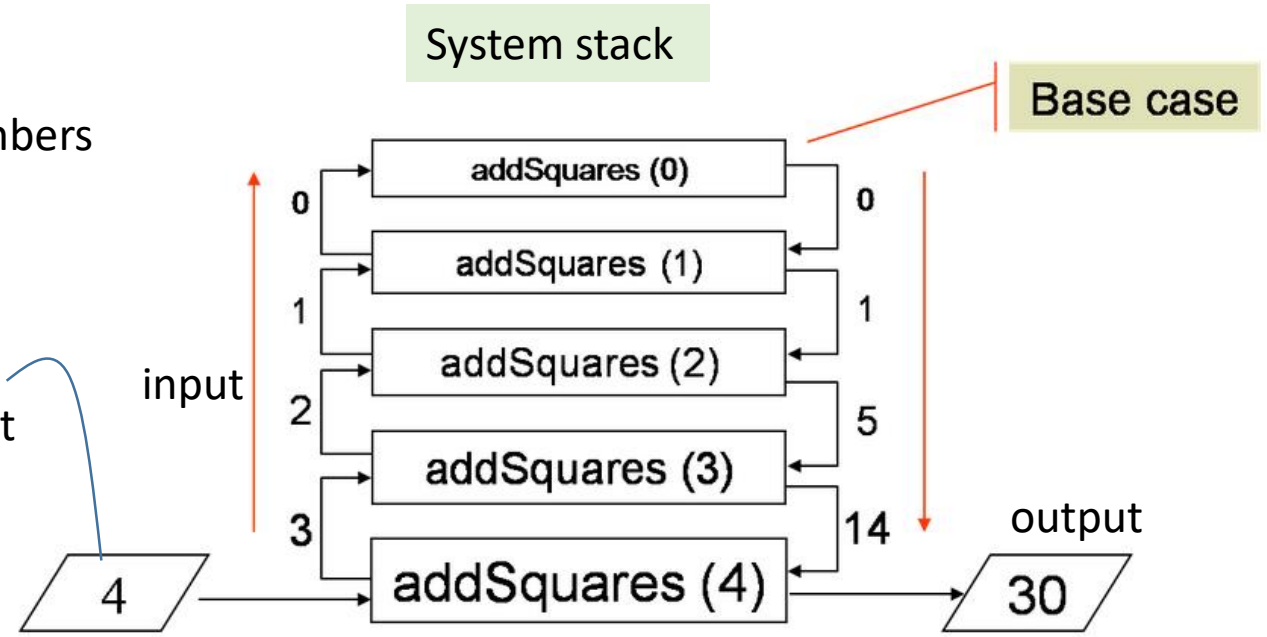
$$N^2 + (N-1)^2 + (N-2)^2 + \cdots + 0^2$$

$$\sum_{i=0}^{N} i^2 = N^2 + \sum_{i=0}^{N-1} i^2 = N^2 + (N-1)^2 + \sum_{i=1}^{N-2} i^2 + 0^2 =$$

```
function [R] = addSquares( N )
    % Recursive function to add squares of natural numbers

    if N==0
        R=0;
    else
        R=N^2+addSquares(N-1)
    end

end
```



Stack

- A *stack* is an abstract data type that serves as a collection of elements, with two principal operations:
- **push**, which adds an element to the collection, and
- **pop**, which removes the most recently added element that was not yet removed.
- The order in which elements come off a stack gives rise to its alternative name:

  LIFO (**last in, first out**).