ITERATIVE VS RECURSIVE METHODS

Before this topic, you should have been 100% familiar with what iterative structure is about and have also done quite a bit exercises with nested control structure.

In this packet, we will talk about:

Sample Comparison ................................................................. 2

Which is Better – iteration or recursion? .......................................... 3

Key design a recursive function ....................................................... 3

Dos ..................................................................................................... 3

Don’ts ............................................................................................... 4

Recursive Programming Exercises .................................................. 5
SAMPLE COMPARISON

Best examples to demonstrate the difference between the two using Factorial and Fibonacci:

Factorial of N:

**HOW IT WORKS**

\[ n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1, \text{ where } n > 0 \quad \text{or} \]

\[ n! = n \cdot (n-1)! \quad \text{where } n \geq 0 \text{ with } 0! = 1 \]

**PERFORM AN ITERATIVE ALGORITHM:**

1. for \( j \) from \( n \) to \( 2 \)
2. \( \text{result} = \text{result} \cdot j \)

**PERFORM A RECURSIVE ALGORITHM:**

3. \( \text{factorial(int } i) \)
4. If \( i == 2 \)
5. \( \text{return } 2 \)
6. \( \text{return } n \cdot \text{factorial}(n-1) \)

Fibonacci of N:

**WITH ITERATIVE METHOD**

\[
\text{long fib (short } n) \\
\quad k1 = k2 = 1; \\
\quad \text{for } j \text{ from } 3 \text{ to } N \\
\quad \text{sum} = k1 + k2; \\
\quad k1 = k2; \\
\quad k2 = \text{sum};
\]

**WITH RECURSIVE METHOD**

\[
\text{fib (int } N) \\
\quad \text{sum} = \text{fib}(N-1) + \text{fib}(N-2)
\]

Two types of recursion

a) Tail Recursion
   State change
   Calling recursive function again
b) Head Recursion
   Calling recursive function again
   State change
**WHICH IS BETTER — ITERATION OR RECURSION?**

— Depends on the implementation - There are known trade-offs.
— Not all algorithms should use recursion. Some can be far worse in recursive algorithms, such as Fibonacci (unless implemented with lookup table for overlapping sub-structure using dynamic programming method.)
— Since it is a stack implementation, need to be far more careful in designing of the recursive function.
— If you are not careful with the program logic, you may miss a base case and go off into an infinite recursion. Once you miss this base case, this can be far worse than running into infinite loop.
— Most advanced algorithms require recursive thinking, implementation.

<table>
<thead>
<tr>
<th>Recursion</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Mathematicians” often prefer recursive approach as it resembles the mathematical formulation</td>
<td>--</td>
</tr>
<tr>
<td>Commonly used in Artificial Intelligence implementation</td>
<td>--</td>
</tr>
<tr>
<td>Highly abstract</td>
<td>Less abstract</td>
</tr>
<tr>
<td>Even good recursive solutions may be much more difficult to design and debug.</td>
<td>Programmers, esp. w/o college CS and computational thinking training, often prefer iterative solutions, as it is far easier to debug.</td>
</tr>
<tr>
<td>Must think in stacks.</td>
<td>Control stays local to loop.. easier to comprehend how data was changed.</td>
</tr>
<tr>
<td>Memory is duplicated for every call.</td>
<td># of iterations, and Memory usage is clear.</td>
</tr>
</tbody>
</table>

**KEY DESIGN A RECURSIVE FUNCTION**

**Dos**

Think in terms of the definition/characteristics of the problem, not the step-by-step process of solving it. For example, Try to express a problem in a single expression (especially in mathematical formulation).

1. Always identify all “base case(s)”; or I like to call the terminating case(s). The simplest instance(s) which do not need recursive calls.
2. Define sub-problems, ie. Instances.
3. Define parameter(s) distinguishes each instance (a stack/sub-task/call).
4. Include the parameters identified in the previous step.
5. Break it up smaller and smaller until chunks until you reach something you know how to deal with. Eventually, you make enough recursive calls that the input reaches a “basis case”.

Example:

Bigger picture: Sum (arr, n)

Smaller chunks: Sum (arr, 1) + arr[1] // there are 2 elements I the array
               Sum (arr, 2) + arr[2] // there are 3 elements I the array
               Sum (arr, 3) + arr[3] // there are 4 elements I the array
               ...
               return sum( &arr[n-1], n-1 ) + arr[n]
               ...

but don’t forget about the base case, i.e. if (n==1) or if (n==2), return the sum value.
6. When the recursive method returns a value, use the return value for (incrementally) building the solution for the task

**Don’ts**

1. Avoid using / updating "global variables" (instance variables). Use local variables if you can. However, of course, only if the “global variables” should be an singleton attribute.

2. Do not modify the parameters that identify the instance of the task. Example:

<table>
<thead>
<tr>
<th>Bad</th>
<th>Good</th>
</tr>
</thead>
</table>
| void sum( int num)  
{  
....  
--num;  
add(num);  
} | void sum( int num)  
{  
....  
add(num-1);  
} |

3. When working with arrays, avoid making copies of the array. Most of the time, this can be done by passing the left- and right- boundaries of the array as additional parameters of the recursive method.

4. Remember recursive stacks can be memory intensive. Therefore, try to minimize creation of any variables which you do not really need.

5. Do not do dynamic allocation, as it can be extremely cost, as you must be the one who free the memory that you allocate.

---

**Recursion Or Not Recursion!!!**

- No substitute for careful thought.
- “Obvious” and “natural” solutions aren’t always practical.
- When you are designing your recursive calls, make sure that at least come up with base case(s).
- Think about memory usage.
- Always think about the trade off between memory usage and elegancy.
- Simplicity and Robust.
RECURSIVE PROGRAMMING EXERCISES

Before you start the following exercises, there are a few things you need to keep in mind:

1) No global.
2) All must be tested as functions.
3) All your tests must allow users to retest without restart the program. Ask for some sentinel value, such as: if user enters -1, it means terminating test.

1. Write a function int baseExp(int base, int exponent) that takes in a base and an exponent and recursively computes base exponent. You are not allowed to use the pow() function!

2. Write a function using recursion “void prtn20(int n)” to print numbers from n to 0.

3. Write a function using recursion “void prtn02n(int n)” to print numbers from 0 to n.

4. Write a function “int countDigits(int n)” to tell the # of digits of a number.

5. Write a function using recursion that takes in a string and copy the reversed order of it into another string. E.g. source string = “Hello!”, target string = “!olleH”. (Do not use any intrinsic string function.)

   // Save the reversed string into a target string. Source must be null-terminated string.
   void reverse (you decide what parameters need to be included in here)

6. Write a recursive function that computes the sum of all numbers from 1 to n, where n is given as parameter.

   //return the sum 1+ 2+ 3+ ...+ n
   int sum(int n)

7. Write a recursive function that finds and returns the minimum element in an array, where the array and its size are given as parameters.

   //return the smallest value in a[
   int findmin(int a[], int n)
8. Write a recursive function that computes and returns the sum of all elements in an array, where the array and its size are given as parameters.

```c
int findsum(int a[], int n)
```

9. Write a recursive function that determines whether an array is a palindrome, where the array and its size are given as parameters.

```c
int ispalindrome(char a[], int n)
```

10. Write a function “void drawTriangle()” to print a “left-justified” 90° triangle of a given height.

    For instance drawTriangle(5) will print:

    ```
    #
    ##
    ###
    ####
    #####
    ```

11. Write a function “void drawTriangleUpsideDown()” of a given height.

    For instance drawTriangleUpsideDown(5) will print:

    ```
    #####
    ####
    ###
    ##
    #
    ```

12. Find the GCD (N, M) using “Euclid Algorithm” - recursively.

13. Write a recursive function to print out bits form of an integer. Ask user for an integer value, and display in:

    a. Hex format (with a space in between each byte)
    b. Binary format (with a space in-between 4 bits)

More next page...
14. Write a recursive function that searches for a target in a sorted array using binary search, where the array, its size and the target are given as parameters.

```c
//binary search a sorted array
int *bsearch (int *list, int nElements, int target )

Or

void *bsearch (void *list, int nElements, int nElementSize, 
    void *compFunction(const void *, const void *))
```

Note: The following two programs This is just for exercise, but not optimal solution to do this in strict form of recursion. When you use a big number over such as 30K (depending on your stack size) will simply blow up your stack and thus program will thus crash. The implementation to make this scalable is not trivial and outside the scope for the time being.

*** May use local static variable in the following two exercises.

15. Write a recursive function that takes in one argument N and computes Fibonacci (N).

16. Write a function using recursion to check if a number n is prime (you have to check whether n is divisible by any number below n). In order to

```c
bool isPrime (int n)
```