ITERATIVE VS RECURSIVE METHODS

Before this topics, 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**

\[ n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 1, \text{ where } n > 0 \]

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

**PERFORM AN ITERATIVE ALGORITHM:**

1. for \( j \) from \( n \) to 2
2. result = result * \( j \)

**PERFORM A RECURSIVE ALGORITHM:**

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

**Fibonacci of N:**

**WITH ITERATIVE METHOD**

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

**WITH RECURSIVE METHOD**

\[
\begin{align*}
\text{fib (int } N) \\
& \text{sum} = \text{fib}(N-1) + \text{fib}(N-2)
\end{align*}
\]

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

   ```c
   Bad
   void sum( int num)
   {
     ....
     --num;
     add(num);
   }
   
   Good
   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

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 prto2n(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(char *target, char *source)
   ```

6. 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).

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

7. 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)
   ```

8. 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)
   ```

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

```c
//returns 1 if a[] is a palindrome, 0 otherwise
int ispalindrome(char a[], int n)
```

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

For instance `drawTriangle(6)` will print:

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

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

For instance `drawTriangleUpsideDown(6)` will print:

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

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

14. Find the GCD (N, M) using “Euclid Algorithm” - recursively.
15. 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 *))
```

Exercises continue next page.
The following exercises are Only for those who know bit-wise operation.

Goal of these exercises is not to teach you how to do bitwise arithmetic, but to have you practiced thinking the attributes / characteristics in using various bits operations.

Write the following functions to print out all the bits of an data field:

16. Perform division with bits operation. No arithmetic operators, “+”, “-”, “/”, “*” is allowed.

In order to come up with the solutions, you need to think about:

1. What attributes does each of the operation possess? Such as, for addition, what operation you need to know whether you need to do carry over? How to carry over?
2. What operation in bitwise operation allowing similar result like “*” and “/”?
3. How to check if it is an odd or even?
4. How to check if a bit is set to 1 or 0?
5. Look for pattern, especially for multiplication and division. For example, for multiplication of 2 numbers, when you use “<” on one number, what you need to consider with the second number?

One way to do division:

Sample 1:

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;1</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;1</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;1</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;1</td>
<td>stop</td>
</tr>
<tr>
<td>1&lt;&lt;2 =</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>34 - 28 =</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

stop
Sample 2:

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>7</td>
<td>(4+2+1)</td>
</tr>
<tr>
<td>$\ll 1$</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>$\ll 1$</td>
<td>28</td>
<td>stop</td>
</tr>
<tr>
<td>$\ll 1$</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>1 $\ll 2$</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>50 - 28</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

22

<table>
<thead>
<tr>
<th>Divisor</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 1$</td>
<td>14</td>
</tr>
<tr>
<td>$\ll 1$</td>
<td>28</td>
</tr>
<tr>
<td>stop</td>
<td></td>
</tr>
<tr>
<td>1 $\ll 1$</td>
<td></td>
</tr>
<tr>
<td>22 - 14</td>
<td>2</td>
</tr>
</tbody>
</table>

8

<table>
<thead>
<tr>
<th>Divisor</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 1$</td>
<td>14</td>
</tr>
<tr>
<td>stop</td>
<td></td>
</tr>
<tr>
<td>1 $\ll 0$</td>
<td>1</td>
</tr>
</tbody>
</table>

See next page for another sample.
Sample 3:

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>1839</td>
<td>115</td>
<td>15</td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>230</td>
<td>8+4+2+1</td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>460</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>920</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>1840</td>
<td>stop</td>
</tr>
<tr>
<td>1839 - 920 = 919</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>919</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>460</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>920</td>
<td>stop</td>
</tr>
<tr>
<td>919 - 460 = 459</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>459</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>460</td>
<td>stop</td>
</tr>
<tr>
<td>459 - 230 = 229</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>229</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>230</td>
<td>stop</td>
</tr>
<tr>
<td>&lt;&lt;1</td>
<td>460</td>
<td>1&lt;&lt;0</td>
</tr>
</tbody>
</table>

17. void prtBits( char *p, int nBytes)

18. void prtBitsEndian( int *p, int nBytes)

    // watch out of the BigEndian vs Little Endian
Computational Thinking and Engineering

Algorithms in C/C++