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

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

Factorial of N:

How it works

$n! = n * (n-1) * (n-1) * (n-2) * (n-3) * .. 1$, where $n > 0$ or

$n! = n * (n-1)!$ where $n >= 0$ 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 *  factorial(n-1)
```

Fibonacci of N:

With Iterative Method

```
long fib (short n)
    k1 = k2 = 1;
    for j from 3 to N
  sum = k1 + k2;
  k1 = k2;
  k2 = sum;
```

With Recursive Method

```
fib (int N)
    sum = fib(N-1) + 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.)
- 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 basis case and go off into an infinite recursion. Once you miss this base case, this can be far worse than running into infinite loop.

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

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".
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:

Bad	Good
void sum(int num)	void sum(int num)

{
    ....
    --num;
    add(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 prt02n(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.

```
//return the sum of all elements in a[]

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

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

```
//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” 90º 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.

```
//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 *))
```

The following exercises are Only for those who know bit-wise operation. 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:

- 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?
- What operation in bitwise operation allowing similar result like "*" and "/"?
- How to check if it is an odd or even?
- How to check if a bit is set to 1 or 0?
- 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:

Dividend		Divisor			Quotient
34		7			4
	<<1	14			
	<<1	28			
	~~<<1~~	~~56~~	stop		
			1<<2 =	4	
			34 - 28 =	6	
6		7	stop		

Sample 2:

Dividend		Divisor			Quotient
50		7			7
	<<1	14			(4+2+1)
	<<1	28			
	~~<<1~~	~~56~~	stop		
			1<<2	4	
			50 - 28 =	22	
22		7			
	<<1	14			
	~~<<1~~	~~28~~	stop		
			1<<1		
			22 - 14 =	2	
8		7			
	~~<<1~~	~~14~~	stop		
			1<<0	1	

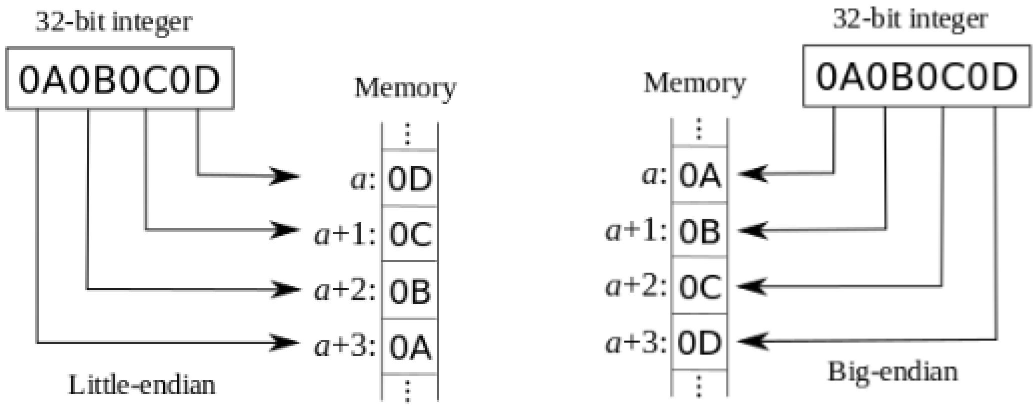
Sample 3:

Dividend		Divisor			Quotient
1839		115			15
	<<1	230			8+4+2+1
	<<1	460			
	<<1	920			
	~~<<1~~	~~1840~~	stop		
			1<<3	8	
			1839 - 920 =	919	
919		115			
	<<1	230			
	<<1	460			
	~~<<1~~	~~920~~	stop		
			1<<2	4	
			919 - 460 =	459	
459		115			
	<<1	230			
	~~<<1~~	~~460~~	stop		
			1<<1	2	
			459 - 230 =	229	
229		115			
	~~<<1~~	~~230~~	stop		
	~~<<1~~	~~460~~	1<<0	1	

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

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

// watch out of the BigEndian vs Little Endian


32-bit integer
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian
32-bit integer
0A0B0C0D
Memory
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian