CSCI-21 Assignment #5 – subroutines. Due 4/25/22

Exercise 1 (using the simple linkage convention):

Declare an array of ten integers. In a loop, prompt for the ten integers (in a subroutine, calling it ten times, passing in the address of each element to store the integer into); then sort the array (in a subroutine, passing in the base of the array and the number of elements in the array); then use a loop to print the sorted array (using a subroutine, passing the value of each integer to print as an argument). The printing subroutine may just print a tab after each number. You may use any simple $O(n^2)$ sort you saw in CSCI-14.

Exercise 2 (Using the stack-based linkage convention)

The value of n!, usually defined as n * (n-1) * ... * 2 * 1, may also be defined as follows (in pseudocode):

```
int fact( int n )
begin
   if( n <= 1 )                    // base case
      return 1
   return n * fact( n-1 )          // general or recursive case
end
```

Implement a recursive function fact() that matches this design. You may not use a loop to calculate this. Write a main program that prompts the user for an integer, calls fact() to get the factorial, then prints the result with reasonable text.

Exercise 3 (Using the frame-based linkage convention)

The Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, ... is defined as follows: choose two non-negative integers (here 0 and 1), then all following values are the sum of the preceding 2 integers. Here the terms are numbered from 0, so fib(7) is 13. This can be defined as follows (in pseudocode):

```
int fib( unsigned int n )
begin
   if( n <= 1
      return n
   a = fib( n-1 )
   b = fib( n-2 )
   return a + b
end
```

Use local variables (in the stack frame) for a and b. You will need to save and restore n (in a local variable) also. Write a program that prompts the user for an integer (you may assume a non-negative entry), calls fib() with than value, then prints a message like this:

fib(n) is f

where n is the entry and f is the result. Test with 0, 1, 2, and several entries not greater than 30. If you're *really* ambitions, try 50 or 60... Have the code ignore the possibility of overflow.

Exercise 4 (Using my "real-world" linkage convention)

Write a recursive subroutine that takes an integer >= 1, and returns the sum of all the squares from 1 to that number: 1^2, 2^2, ... , $(n-1)^2$, n^2.

The algorithm looks like this:

```
declare local variable to hold square of n
if( n <= 1 )
    return 1
end if
calculate n squared and save it in the local variable
call subroutine with n-1
add return value to saved value and return that value
```

Write a main routine that prompts the user for an integer, keeps that integer in a safe register, calls the sum-of-squares subroutine, then (afterward) prints a message like this:

```
SumInts of n is m
```

where n is the user's entry and m is the value returned from the function.

You may not use $a registers for argument passing: the value must be a parameter passed as a variable on the stack. Use a local variable (on the stack) to save the value of the parameter squared, which must be calculated before the recursive subroutine call. Careful use of registers will minimize the number of registers that must be saved and restored at each subroutine call. You must print all of the result message after the subroutine call: you may not print part of it, call the subroutine, then print the rest of the message. Test with 1, 2, 3, and several values up to 100. Have the code ignore possible overflow in the multiplication and addition operations needed for the subroutine.

Note: for all of these exercises, you must follow the calling conventions as stated. You may not, for example, determine that you are in the base case of the recursive subroutine and then skip the callee prolog and callee epilog since they aren't really needed for that. The compiler will not short-cut the convention, so don't do that yourself.