Recursion Lab

You should not use loops when writing any of these functions, unless otherwise specified.

1. Suppose we want to write a recursive function that takes an argument n, that adds up all the numbers between 1 and n, inclusive. This function has the signature: `int sum(int n)`. Define a recursive formulation below:

   Base case: if n = _____ , then sum(n) = _____

   Recursive case: if n > _____ , then sum(n) = _____ + _____

   Translate your recursive formulation into C++. Test your code in main().

2. Suppose we want to write a recursive function that takes an argument n, that counts all the prime numbers between 2 and n, inclusive. Signature is `int count_primes(int n)`. Assume you have access to another function, `is_prime(n)`, that returns a boolean that states whether n is a prime number or not. Define a recursive formulation:

   Base case: if n = _____ , then count_primes(n) = _____

   Recursive case A: if n > ____ AND _____, then count_primes(n) = _____ + _____

   Recursive case B: if n > ____ AND _____, then count_primes(n) = _____ + _____

   Translate your recursive formulation into C++. *is_prime is already written for you*.
   Test your code in main().

3. Write a recursive function to compute a^b, (a raised to the b power) where a and b are non-negative integers. The function's signature is `long long power(a, b)`. (Obviously, you may not use the built-in pow function to solve this.)

   Base case: if b = _____ , then power(a, b) = _____

   Recursive case(s):
   (up to you!)

   Translate your recursive formulation into C++. Test your code in main().

   Before moving on, think about how many total multiplications your code does. Can lower the number of multiplications? Hint: to compute (for instance) 2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2, you don't actually need seven multiplications.

   (turn page over)

4. Write a function to test if a string is a palindrome (reads the same forwards and backwards).

   `bool is_pal(string s):`

   Base case: if s.length() < _____ , then is_pal(s) = _____

   Recursive case A: if s.length() >= _____ AND _____,

   then is_pal(s) = _____

   Recursive case B: if s.length() >= _____ AND _____,

   then is_pal(s) = _____

5. Suppose we want to write a recursive function to turn an integer into a *simplified* Roman numeral. *Simplified* in this case means we will ignore the subtraction rules. Roman numerals use seven symbols: I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000. Read the Wikipedia page for Roman numerals if you need a refresher on how to interpret them.

   Write a function `string int_to_roman(int n)` that returns (not prints) an int as a Roman numeral.

6. Write a recursive function to find the maximum element in a vector of integers. You've done this with a loop many times, but now you will write a recursive version.

   Hint: The "natural" recursive formulation involves calling the function recursively on slices of the original list that continue getting smaller:

   Base case: the maximum element in a vector of size 1 is the lone element in the vector; that is, vec[0]
   Recursive case: the maximum element in a vector of size > 1 is the larger of vec[0] and the maximum element in vec[1:size] *(using Python slice syntax here)*

   Because in C++ there is no convenient "slice" operator like there is in Python, we must be a little clever to work around this. Instead of slicing the list over and over, we have our max function take two extra arguments, that specify our current slice into the vector. That is, when you call this function recursively, vec never changes, only start & end change.

   `int max(const vector<int> & vec, int start, int end)`

   Base case: if start == end, then max = _____

   Recursive case: otherwise, then, if _____ > _____, then max = _____

   if _____ <= _____, then max = _____