Lab 4 – Recursive Thinking and Problem Solving

Base Case
solve the simplest version(s) of the problem directly

General Case
(a) make the problem slightly smaller (i.e., closer to the base case)
(b) let the recursion (i.e., the wizard) "magically" solve the smaller version of the problem for you
(c) use the result of (b) to help you solve the original version of the problem

1. Write the function **`turn-to-frogs`**, which takes a list of numbers and returns a new list with all of the numbers replaced by the symbol `frog`. Your function should behave as shown in the examples below:

```
(turn-to-frogs '(2 3 4 5))    → (frog frog frog frog)
(turn-to-frogs '(1 2 3 4 5))  → (frog frog frog frog frog)
(turn-to-frogs '())           → ()
(turn-to-frogs '(13 42 99))   → (frog frog frog)
```

2. Write the function **`create-frog-pile`**, which takes a number *n* as input and returns a new list containing *n* `frog` symbols.

```
(create-frog-pile 5) → (frog frog frog frog frog)
(create-frog-pile 0) → ()
(create-frog-pile 1) → (frog)
```

3. Write the function **`count-frogs`**, which takes a list of `frog` symbols and counts how many `frog` symbols are in the list. For this exercise, you may assume that the list will either be empty, or will contain only frogs.

```
(count-frogs '(frog frog frog)) → 3
(count-frogs '()) → 0
(count-frogs '(frog)) → 1
```

4. Write the function **`count-just-frogs`**, which takes a mixed list of frogs, wizards, trolls, or other strange creatures, and counts just the frogs. Hint: add an extra condition to the cond or if-expression that you wrote for your `count-frogs` function to handle the case when the first creature in the list is *not* a frog.

```
(count-just-frogs '(frog frog wizard troll frog orc hobbit)) → 3
(count-just-frogs '()) → 0
(count-just-frogs '(wizard wizard wizard frog)) → 1
(count-just-frogs '(troll elf dwarf)) → 0
```

5. Write the function **`count`**, which takes a symbol and a list of symbols as input, and counts the number of times the symbol appears in the list. Hint: this function is very similar to the `count-just-frogs` function, except that it takes *two* input parameters instead of just one, where the first parameter can be any symbol.

```
(count 'troll '(frog frog wizard troll frog orc hobbit)) → 1
(count 'wizard '(wizard wizard wizard frog)) → 3
(count 'frog '(troll elf dwarf)) → 0
(count 'elf '(elf elf)) → 2
```

6. Write the function **`factorial`**, which takes a number *n* as input and returns the factorial of *n*. The factorial of *n* is written *n*!, and is the product of $n \times n{-}1 \times n{-}2 \times \ldots \times 3 \times 2 \times 1$. Or in other words, the factorial of *n* is equal to *n* times the factorial of *n*–1. By definition, the factorial of 0 is 1.

```
(factorial 0)  →  1
(factorial 4)  →  24
(factorial 5)  →  120
```

7. Using your `factorial` function as a guide, write the function `(`**`power`** *`base exponent`*`)` that raises a number *base* to the power of *exponent*. By definition, any *base* raised to the power of 0 is 1. For example, `(power 2 0)` should give 1, and `(power 2 5)` should give 32. Hint: if the wizard told you that `(power 2 4)` gives 16, how could you use the wizard's answer to compute the answer for `(power 2 5)`? You are not allowed to use Scheme's built-in `expt` function for this exercise.

```
(power 2 0)   →  1
(power 2 5)   →  32
(power 10 3)  →  1000
(power 4 1)   →  4
(power 3 5)   →  243
```

8. Write the function **`sum-of-first`**, which takes a number *n* as input and adds up all of the numbers from 1 to *n*. For example, `(sum-of-first 5)` should give 1 + 2 + 3 + 4 + 5 = 15, and `(sum-of-first 100)` should give 5050. Notice that this problem is different from the `addup` function we wrote in class, which took a *list of numbers* and added them all up, whereas here the input to the `sum-of-first` function is just a single number *n*. Hint: if the wizard tells you that `(sum-of-first 99)` gives 4950, how could you use the wizard's answer to compute the answer for `(sum-of-first 100)`?

```
(sum-of-first 4)  →  10
(sum-of-first 5)  →  15
(sum-of-first 100)  →  5050
(sum-of-first 500)  →  125250
```

9. Write the function `(`**`remove-from-front`** *`n input-list`*`)`, which takes a number *n* and a list of symbols as input, and removes the first *n* symbols from the list. For this exercise, you may assume that there will always be at least *n* symbols in the input list. Hint: to remove three symbols from the front of a list, remove the first one yourself and then ask the wizard to remove two more symbols from the front of the remaining list for you.

```
(remove-from-front 3 '(a b c d e))  →  (d e)
(remove-from-front 2 '(b c d e))  →  (d e)
(remove-from-front 0 '(frog frog frog))  →  (frog frog frog)
(remove-from-front 4 '(john paul george ringo))  →  ()
```

10. Write the function `(`**`get-symbol`** *`n input-list`*`)`, which takes a number *n* and a list of symbols as input, and retrieves the symbol at position *n* in the list, counting from 0. Hint: to retrieve the symbol at position 5, make the list shorter by one symbol and then ask the wizard to retrieve the symbol at position 4 in the shorter list.

```
(get-symbol 0 '(a b c d e f))  →  a
(get-symbol 5 '(a b c d e f))  →  f
(get-symbol 4 '(b c d e f))    →  f
(get-symbol 2 '(first second third))  →  third
```

11. Write the function **`double-each`**, which takes a list of numbers and returns a new list containing each number from the input list multiplied by 2.

```
(double-each '(1 2 3))  →  (2 4 6)
(double-each '())  →  ()
(double-each '(5 10 15 20 25))  →  (10 20 30 40 50)
```

12. Write the function **`add-to-end`**, which takes a symbol and a list as input, and returns a new list with the symbol added to the end of the list. Hint: if the input list is empty, your function should return the symbol in a list by itself.

```
(add-to-end 'frog '()) → (frog)
(add-to-end 'captain '(aye aye)) → (aye aye captain)
(add-to-end 'cream '(big frozen blocks of ice)) → (big frozen blocks of ice cream)
```

13. Write the function **`get-last`**, which takes a list of symbols as input, and returns the last symbol in the list. For this exercise, you may assume that the input list will always contain at least one symbol.

```
(get-last '(once upon a time))  →  time
(get-last '(apple))  →  apple
(get-last '(the rain in Spain stays mainly in the plain))  →  plain
```

14. Write the function **`remove-last`**, which takes a list of symbols as input, and returns a new list with the last symbol removed. For this exercise, you may assume that the input list will always contain at least one symbol.

```
(remove-last '(apple))  →  ()
(remove-last '(we all scream for ice cream))  →  (we all scream for ice)
(remove-last '(red fish blue fish))  →  (red fish blue)
```

15. Write the function **`replace-last`**, which takes a symbol and a list of symbols as input, and returns a new list with the last symbol in the list replaced by the input symbol.

```
(replace-last 'water '(we all scream for ice cream))  →  (we all scream for ice water)
(replace-last 'gold '(silver))  →  (gold)
(replace-last 'frog '(red fish blue fish))  →  (red fish blue frog)
```

16. Write the function **`range`**, which takes two numbers *start* and *end* as input, and returns a new list of numbers in sequence from *start* to *end*. If *start* > *end*, then an empty list is returned.

```
(range 1 8)  →  (1 2 3 4 5 6 7 8)
(range 2 8)  →  (2 3 4 5 6 7 8)
(range 8 8)  →  (8)
(range 9 8)  →  ()
```