Contents

1 Basic Test Results

```
1  Starting tests...
2  Mon Dec  7 11:19:49 IST 2015
3  e791840f6d8a042adcc580362f0f921b8d25b1e7  -
4
5
6  Archive:  /tmp/bodek.rfUboC/intro2cs/ex7/elinorperl/presubmission/submission
7    inflating: src/ex7.py
8    inflating: src/README
9
10 Testing README...
11 Done testing README...
12
13 Running presubmit tests...
14 40 passed tests out of 40
15 result_code    ex7    40    1
16 Done running presubmit tests
17
18 Tests completed
19
20 Additional notes:
21
22 There will be additional tests which will not be published in advance.
```

2 README

```
elinorperl
329577464
Elinor Perl

I discussed the exercise with Talya Adams, Or Sharbat, Bnaya Pelet

========================================
=  README for ex7: Recursive Functions  =
========================================


==================
=  Description:  =
==================

In this exercise, I worked with different types of recursive functions, linear
and non-linear. Functions that were implemented were ones that worked recursively.
This exercise incorporated use of helper function and loops to implement the
recusive function given.

======================
=  Special Comments  =
======================

I used stackoverflow.com
```

3 ex7.py

```
def print_to_n(n):
    """
     Prints out the natural numbers from one until the designated number that
      was picked, otherwise printing an empty string.
    """
    if n > 0:
        if n == 1:
            print(1)
        else:
            print_to_n(n-1)
            print(n)

def print_reversed_n(n):
    """
    Prints out natural numbers, starting from the designated number that was
    chosen, counting down until 1, otherwise printing out an empty string.
    """
    if n > 0:
        if n == 1:
            print(1)
        else:
            print(n)
            print_reversed_n(n-1)


def has_divisor_smaller_than(n, i):
    """ A helping function, set as a Boolean type of function, checking if the
    first number set has a smaller divisor than the second input and
    continuing to a smalled secondary number until it reaches 1.
    """
    if i == 1:
        return True
    if n % i == 0:
        return False
    else:
        return has_divisor_smaller_than(n, i-1)


def is_prime(n):
    """ Using the helpful function "has_divisor_smaller_than", I checked
    if the number that was input is a prime number, by checking if the first
    number could be divided by anything smaller than itself.
    """
    if n <= 1:
        return False
    else:
        return has_divisor_smaller_than(n, n-1)



def find_divisors(n, i, divisor_list):
    """
    A helpful function, creating a list of all the numbers that are divisors
    of the first input number, starting from the second input and creating
    a list of those numbers.
    """
    if i > n:
        return divisor_list
    elif n % i == 0:
```

```
60          divisor_list.append(i)
61      return find_divisors(n, i + 1, divisor_list)
62
63
64  def divisors(n):
65      """
66      I created a list of the divisors, using the helpful function "find_
67      divisors", starting from 1, until the absolute value of the number that
68      was input.
69      """
70      divisor_list = []
71      return find_divisors(abs(n), 1, divisor_list)
72
73
74  def factorial(n):
75      """
76      Factorial is a helpful function that returns the factorial of the given
77      number.
78      """
79      if n == 0:
80          return 1
81      else:
82          return n*factorial(n-1)
83
84
85  def exp_n_x(n,x):
86      """
87      Using the helpful function "Factorial" and our current one recursively,
88      I forged a Sigma sum with the two numbers input.
89      """
90      if n == 0:
91          return x**n/factorial(n)
92      else:
93          return x**n/factorial(n) + exp_n_x(n-1, x)
94
95
96  def play_hanoi(hanoi, n, src, dest, temp):
97      """
98      Using the graphics given to us, I solved the hanoi towers recursively,
99      by recalling the original function and moving the disk, and recalling the
100      function.
101     """
102     if n >= 1:
103         play_hanoi(hanoi,n-1, src, temp, dest)
104         hanoi.move(src, dest)
105         play_hanoi(hanoi, n-1, temp, dest, src)
106
107
108 def print_binary_sequences_with_prefix(prefix, n):
109     """
110     A help function, which returns a series of 0's and 1's according to the
111     length that was input. The function returns all the possible patterns
112     in regards to the length.
113     """
114     if len(prefix) == n:
115         print(prefix)
116     else:
117         copied_prefix = prefix[:]
118         print_binary_sequences_with_prefix(copied_prefix + "0", n)
119         print_binary_sequences_with_prefix(copied_prefix + "1", n)
120
121 def print_binary_sequences(n):
122     """
123     Using the recursive helper function "print_binary_sequences_with_prefix",
124     I called it into my current function to get the necessary binary sequence.
125     """
126     if n >= 0:
127         prefix = ""
```

```
        print_binary_sequences_with_prefix(prefix, n)


def print_sequences_with_sequence(char_list, n, sequence):
    """
    A helping function that recursively prints out each sequence once it's the
     length that was input, repeating itself to get each combination from the
     char_list that was input.
    """
    if len(sequence) == n:
        print(sequence)
    else:
        for char in char_list:
            copied_sequence = sequence[:]
            print_sequences_with_sequence(char_list, n, copied_sequence + char)


def print_sequences(char_list, n):
    """
    Using the helping function "print_sequences_with_sequence", I created an
    empty sequence string and called upon the helping function with my givens
    from this function.
    """
    sequence = ""
    print_sequences_with_sequence(char_list, n, sequence)


def no_repetition_helper(char_list, n, sequence):
    """
    This helper function operates similarly to the former, where according to
    the input sequences, the function returns every combination, but this time
    without repetition of any letter. This was done by deleting and
    reinserting the characters, as to not let the recursive function re-use
    the character that was already used.
    """
    if len(sequence) == n:
        print(sequence)
    else:
        for i in range(len(char_list)):
            temp_char = char_list[i]
            del char_list[i]
            no_repetition_helper(char_list, n, sequence + temp_char)
            char_list.insert(i, temp_char)


def print_no_repetition_sequences(char_list, n):
    """This function calls onto it's helping function "no_repetition_helper"
    and uses an empty sequence string and applies it to the helper in order
    to get each combination of sequences from the char_list.
    """
    sequence = ""
    no_repetition_helper(char_list, n, sequence)


def no_repetition_helper_list(char_list, n, sequence, sequence_list):
    """
    This helping function works similarly to the above function, creating
    each combination of sequences (up until the length of n) from the list
    that was given, but in this function, a list is formed from the
    combination of sequences.
    """
    if len(sequence) == n:
        sequence_list.append(sequence)
    else:
        for i in range(len(char_list)):
            temp_char = char_list[i]
            del char_list[i]
            no_repetition_helper_list(char_list, n, sequence + temp_char,\
```

```
                                        sequence_list)
                char_list.insert(i, temp_char)
    return sequence_list


def no_repetition_sequences_list(char_list, n):
    """
    This combination calls upon the helper function "no_repetition_helper_
    list". I created an empty string and a new list to apply to the helper
    function and get that results.
    """
    sequence = ""
    sequence_list = []
    return no_repetition_helper_list(char_list, n, sequence, sequence_list)
```