Contents

1 Basic Test Results

```
Starting tests...
Fri 28 Jun 2024 00:12:51 IDT
563e6d84efff202eeb5ccefc7524047eb62f3fa0  -


Archive:  /tmp/bodek.51vlmx5k/intro2cs2/ex7/daniel.rez/presubmission/submission
  inflating: src/ex7.py


Running presubmit code tests...
9 passed tests out of 9 in test set named 'ex7'.
result_code    ex7    9    1
Done running presubmit code tests

Finished running the presubmit tests

Additional notes:

The presubmit tests do not check if you used functions or operators you are not
supposed to use.

Make sure to thoroughly test your code.

```

2 ex7.py

```python
##################################################################
# FILE : image_editor.py
# WRITER : daniel_riazanov , daniel.rez , 336119300
# EXERCISE : intro2cs ex7 2024
# DESCRIPTION: Practising recursions
# STUDENTS I DISCUSSED THE EXERCISE WITH: None
# WEB PAGES I USED: None
# NOTES: In order to understand recursion, we need to be those, who already understand recursion
##################################################################

###############################################################################
#                                   Imports                                   #
###############################################################################
from typing import *
import ex7_helper
# Global type definition
N = ex7_helper.N


###############################################################################
#                              exercise function 1                            #
###############################################################################
def mult(x: N, y: int) -> N:
    """
      Multiplies two numbers using only recursion and  helper add & subtract_1 func
      Principle: adding x to x y times

      :param x: The first number to multiply.
      :type x: N
      :param y: The second number to multiply.
      :type y: int
      :return: The product of x and y.
      :rtype: N
    """
    # Base case
    if y == 0:
        return 0
    # If not reaches base case, add x and x y times each time reducing y by 1.
    else:
        return ex7_helper.add(x, mult(x, ex7_helper.subtract_1(y)))


###############################################################################
#                              exercise function 2                            #
###############################################################################
def is_even(n: int) -> bool:
    """
     Checks if a number is even using recursion and subtract_1 func.
     Principle: If by subtracting 2 we reached zero num is even, otherwise we reached 1 and num is odd.

     :param n: The number to check.
     :type n: int
     :return: True if the number is even, False otherwise.
     :rtype: bool
    """
    # Base case
    if n == 0:
        return True
    # Base case
    elif n == 1:
        return False
    # Until base case reached, performs n-2 recursively
    else:
        return is_even(ex7_helper.subtract_1(ex7_helper.subtract_1(n)))


#############################################################################
#                              exercise function 3                          #
#############################################################################
def log_mult(x: N, y: int) -> N:
    """
     Multiplies two numbers using logarithmic recursion and helper add, divide_by_2, and is_odd functions.
     Principle: Reduces the problem size by dividing y by 2 at each step, similar to exponentiation by squaring.

     :param x: The first number to multiply.
     :type x: N
     :param y: The second number to multiply.
     :type y: int
     :return: The product of x and y.
     :rtype: N
    """
    # Base case: if y is 0, return 0 as anything multiplied by 0 is 0
    if y == 0:
        return 0
    # Recursive case: divide y by 2 and call log_mult recursively

    temp = log_mult(x, ex7_helper.divide_by_2(y))
    # If y is even, adds temp to itself
    if not ex7_helper.is_odd(y):
        return ex7_helper.add(temp, temp)
    else:
        # If y is odd, adds x to the double of temp
        return ex7_helper.add(x, ex7_helper.add(temp, temp))


#############################################################################
#                              exercise function 4                          #
#############################################################################
def power(b: int, n: int) -> int:
    """
     Calculates b raised to the power of n using recursion and helper functions divide_by_2, log_mult, and is_odd.
     Principle: Reduces the exponentiation problem size by dividing n by 2, similar to exponentiation by squaring.
     Time Complexity: O(log(n))

     :param b: The base number.
     :type b: int
     :param n: The exponent.
     :type n: int
     :return: The result of b raised to the power of n.
     :rtype: int
    """
    # Base case: any number to the power of 0 is 1
    if n == 0:
        return 1

    # Recursive case: divides n by 2 and call power recursively
    half_power = power(b, ex7_helper.divide_by_2(n))
    # Squares the result of half_power
    half_power_squared = log_mult(half_power, half_power)

    # If n is odd, multiplies the squared result by b
    if ex7_helper.is_odd(n):
        return mult(half_power_squared, b)
    else:
        # If n is even, returns the squared result
        return half_power_squared

```

```python

def is_power_helper(b: int, x: int, low: int, high: int) -> bool:
    """
       Helper function to determine if x is a power of b using binary search.
       Principle: Uses binary search to efficiently find the exponent n such that b^n = x.
       Time Complexity: O(log(x))

       :param b: The base number.
       :type b: int
       :param x: The number to check.
       :type x: int
       :param low: The lower bound of the search range.
       :type low: int
       :param high: The upper bound of the search range.
       :type high: int
       :return: True if x is a power of b, False otherwise.
       :rtype: bool
    """

    # Base case: if low exceeds high, x is not a power of b
    if low > high:
        return False

    # Calculates the midpoint of the current range
    mid = ex7_helper.divide_by_2(low + high)
    # Calculates b raised to the power of mid
    current_power = power(b, mid)

    # Checks if current_power matches x
    if current_power == x:
        return True
    elif current_power < x:
        # If current_power is less than x, searches the upper half
        return is_power_helper(b, x, ex7_helper.add(mid, 1), high)
    else:
        # If current_power is greater than x, searches the lower half
        return is_power_helper(b, x, low, ex7_helper.subtract_1(mid))


def is_power(b: int, x: int) -> bool:
    """
       Determines if b^n = x for some integer n using recursion and helper function is_power_helper.
       Principle: Uses binary search to find the exponent n such that b^n = x.
       Time Complexity: O(log(b) * log(x))

       :param b: The base number.
       :type b: int
       :param x: The number to check.
       :type x: int
       :return: True if b^n equals x for some integer n, False otherwise.
       :rtype: bool
    """
    # Special cases: handle b = 0 and b = 1 separately
    if b == 0:
        return x == 0
    if b == 1:
        return x == 1

    # Uses the helper function to check if x is a power of b
    return is_power_helper(b, x, 1, x)


################################################################################
#                               exercise function 5                            #
################################################################################
def reverse_helper(s: str, index: int, reversed_s: str) -> str:
    """
    Helper function to reverse a string using recursion and the helper function append_to_end.
    Principle: Constructs the reversed string by appending characters from the end of the original string to new string.

       :param s: The original string.
       :type s: str
       :param index: The current index in the original string being processed.
       :type index: int
       :param reversed_s: The reversed string being constructed.
       :type reversed_s: str
       :return: The reversed string.
       :rtype: str
    """
    # Base case: if index is -1, returns the reversed string constructed so far
    if index == -1:
        return reversed_s
    # Recursive case: appends the current character to the reversed string and processes the next character
    return reverse_helper(s, index - 1, ex7_helper.append_to_end(reversed_s, s[index]))


def reverse(s: str) -> str:
    """
        Reverses a string using recursion and the helper function reverse_helper.
        Principle: Uses a helper function to construct the reversed string by processing characters from the end of the
        original string.

        :param s: The string to reverse.
        :type s: str
        :return: The reversed string.
        :rtype: str
    """
    # Calls the helper function starting with the last index of the string and an empty reversed string
    return reverse_helper(s, len(s) - 1, "")


################################################################################
#                               exercise function 6                            #
################################################################################
def play_hanoi(Hanoi: Any, n: int, src: Any, dest: Any, temp: Any):
    """
    Recursive function to solve Tower of Hanoi puzzle.

    :param Hanoi: The game engine object that handles the game state.
    :param n: Number of disks to move.
    :param src: Source tower object.
    :param dest: Destination tower object.
    :param temp: Temporary tower object (third tower) often serves as temp place for swapping discs
    """
    if n <= 0:
        return

    # Move n-1 disks from source to temporary tower
    play_hanoi(Hanoi, n - 1, src, temp, dest)

    # Move the n-th disk from source to destination tower
    Hanoi.move(src, dest)

    # Move the n-1 disks from temporary tower to destination tower
    play_hanoi(Hanoi, n - 1, temp, dest, src)


################################################################################
#                               exercise function 7                            #
################################################################################
def number_of_ones(n: int) -> int:
    """
      Counts the number of times the digit '1' appears in all numbers from 1 to n.
      Principle: Recursively counts '1's in the current number and adds it to the count from previous numbers.

      :param n: The upper limit of the range to count '1's in.
      :type n: int
      :return: The count of '1's in all numbers from 1 to n.
      :rtype: int
    """

    def count_ones_in_current_num(current_num: int) -> int:
        """
          Counts the number of times the digit '1' appears in a single number.
          Principle: Recursively checks each digit of the number, counting occurrences of '1'.

          :param current_num: The number in which to count the digit '1'.
          :type current_num: int
          :return: The count of '1's in the number.
          :rtype: int
        """
        # Base case: if the number is 0, there are no '1's
        if current_num == 0:
            return 0
        # Recursive case: checks the last digit and continues with the rest of the number
        return (1 if current_num % 10 == 1 else 0) + count_ones_in_current_num(current_num // 10)

    # Base case: if n is 0, there are no '1's to count
    if n == 0:
        return 0
    # Recursive case: counts '1's in the current number and adds it to the count from previous numbers
    return count_ones_in_current_num(n) + number_of_ones((n - 1))


##############################################################################
#                              exercise function 8                           #
##############################################################################
def compare_2d_lists(l1: List[List[int]], l2: List[List[int]]) -> bool:
    """
       Compares two 2D lists for equality using recursion and helper functions.
       Principle: Recursively compares the structure and elements of the 2D lists. The problem is divided to 3
       sub-problems (separate helper function for each):
       1. comparing elements in list
       2. comparing inner lists utilizing 1
       3. comparing outer lists utilizing 2

       :param l1: The first 2D list.
       :type l1: List[List[int]]
       :param l2: The second 2D list.
       :type l2: List[List[int]]
       :return: True if the 2D lists are equal, False otherwise.
       :rtype: bool
    """

    def compare_members(inner1: List[int], inner2: List[int], index: int) -> bool:
        """
            Compares members of two inner lists at a specific index using recursion.
            Principle: Recursively compares each element of the two lists to check for equality.

            :param inner1: The first inner list.
            :type inner1: List[int]
            :param inner2: The second inner list.
            :type inner2: List[int]
            :param index: The current index in the inner lists being compared.
            :type index: int
            :return: True if all elements in the inner lists are equal, False otherwise.
            :rtype: bool
        """
        # Base case: if the end of the list is reached, the lists are equal
        if index == len(inner1):
            return True
        # Checks if the current elements are not equal
        if inner1[index] != inner2[index]:
            return False
        # Recursive case: compares the next elements in the lists
        return compare_members(inner1,inner2,index+1)

    def compare_inner_lists(inner1: List[int], inner2: List[int]) -> bool:
        """
            Compares two inner lists for equality using recursion.
            Principle: Checks the lengths of the lists first, then compares each element using a helper function.

            :param inner1: The first inner list.
            :type inner1: List[int]
            :param inner2: The second inner list.
            :type inner2: List[int]
            :return: True if the inner lists are equal, False otherwise.
            :rtype: bool
        """
        # Checks if the lengths of the lists are different
        if len(inner1) != len(inner2):
            return False
        # Compares the elements of the inner lists
        return compare_members(inner1, inner2,0 )

    def compare_outer_lists(outer1: List[List[int]], outer2: List[List[int]], index: int) -> bool:
        """
           Compares two outer lists of lists for equality using recursion.
           Principle: Checks the lengths of the outer lists first, then compares each pair of inner lists using a helper func
           Time Complexity: O(n * m) where n is the length of the outer lists and m is the average length of the inner lists.

           :param outer1: The first outer list of lists.
           :type outer1: List[List[int]]
           :param outer2: The second outer list of lists.
           :type outer2: List[List[int]]
           :param index: The current index in the outer lists being compared.
           :type index: int
           :return: True if the outer lists are equal, False otherwise.
           :rtype: bool
        """
        # Checks if the lengths of the outer lists are different
        if len(outer1) != len(outer2):
            return False
        # Base case: if the end of the outer list is reached, the lists are equal
        if index == len(outer1):
            return True
        # Checks if the current pair of inner lists are not equal
        if not compare_inner_lists(outer1[index], outer2[index]):
            return False
        # Recursive case: compares the next pair of inner lists in the outer lists
        return compare_outer_lists(outer1, outer2, index + 1)

    # Uses the helper function to compare the outer lists
    return compare_outer_lists(l1, l2, 0)


##############################################################################
#                              exercise function 9                           #
##############################################################################
def magic_list(n: int) -> List[Any]:
    """
      Generates a list of lists where each list is a deep copy and follows a pattern similar to an arithmetic sequence.

      The pattern is as follows:
      - For n=0, returns []
      - For n=1, returns [[]]
      - For n=2, returns [[], [[]]]
      - For n=3, returns [[], [[]], [[], [[]]]]
      - And so on...

      :param n: A non-negative integer representing the level of nested lists to generate.
      :type n: int
      :return: A nested list structure following the described pattern.
      :rtype: List[Any]
      """

    if n == 0:
        return []
    # Base case: n=0, returns an empty list

    if n == 1:
        # Base case: n=1, returns a list containing an empty list
        # We need at leas 2 base cases to clarify the pattern
        return [[]]

    #  For n > 1:  defines a nested helper function build_list to construct the list recursively.
    def build_list(current: int) -> List[Any]:
        """
          Recursively builds the list structure from the bottom up.

          :param current: The current level of nested lists being constructed.
          :type current: int
          :return: The constructed list for the current level.
          :rtype: List[Any]
        """

        if current == 0:
            # Base case for recursion: if current is 0 (reached bottom), returns an empty list (which is a1 in sequence)
            return []
        # Build the rest of the list recursively (adding to base case)
        rest_of_list = build_list(current - 1)
        # In this step we're generating a new list for the current level by calling magic_list(current - 1)
        new_list = magic_list(current - 1)
        # Returns the combined list (ensuring deep copy)
        return rest_of_list + [new_list]

    # Starts the recursive construction from level n
    return build_list(n)
```