Assignment 1

THERE ARE 3 PARTS TO THIS ASSIGNMENT – MAKE SURE TO DO ALL THREE!

Part1: Generating permutations in lexicographic order.

Goal: Write a Python program to generate all the permutations of the characters in a string. This will give you a chance to review some simple Python constructs, i.e. Strings and Lists and solidify your understanding of recursion.

Your program **must** meet the following specification. You are to write a Python function **perm_gen_lex** that:

- Takes a string as a single input argument. You may assume the input string consists 0 or more unique lower-case letters in alphabetical order.
- Returns a Python **list** of strings where each string represents a permutation of the input string. The list of permutations must be in lexicographic order (dictionary order). Note: If you follow the pseudo code below, your list will be constructed such that this condition will be met. **Do not sort the list.**
- Is well structured, commented, and easy to read. Contains a docstring explaining its purpose.
- Is recursive and follows the pseudo code below.

```
Argument:      ‘’
Returns:       []

Argument:      ‘a’
Returns:       [‘a’]

Argument:      ‘abc’
Returns:       [‘abc’, ‘acb’, ‘bac’, ‘bca’, ‘cab’, ‘cba’]
```

Pseudo code for a recursive algorithm to generate permutations in lexicographic order.

You must follow this pseudo code.

```
For each character in the input string:
  Form a simpler string by removing the character from the input string
  Generate all permutations of the simpler string recursively (i.e. call the
    perm_gen_lex function with the simpler string)
  Add the removed character to the front of each permutation of the simpler string, and
    add the resulting permutation to the list
```

Note: For a string with n characters, you program will return a list contain n! strings. Note that n! grows very quickly. For example, 15! is roughly $1.3*10^{12}$. Thus it is probably not a good idea to test your program with long strings if you plan on turning the assignment in ‘on’ time.

Part2: Base conversion

One algorithm for converting a base 10 number to base b involves repeated division by the base b. Initially one divides the number by b. The remainder from this division is the units digit (the rightmost digit) in the base b representation of the number (it is the part of the number that contains no powers of b). The quotient is then divided by b on the next iteration. The remainder from this division gives the next base b digit from the right. The quotient from this division is used in the next iteration. The algorithm stops when the quotient is 0. Note that at each iteration the remainder from the division is the next base b digit from the right—that is, this algorithm finds the digits for the base b number in reverse order.

Here is an example for converting 30 to base 4:

```
        quotient remainder
        -------- ---------
30/4       7        2
7/4        1        3
1/4        0        1
```

The answer is read bottom to top in the remainder column, so 30 (base 10) = 132 (base 4).

Think about how this is recursive in nature:

If you want to convert x (30 in our example) to base b (4 in our example), the rightmost digit is the remainder x % b. To get the rest of the digits, you perform the same process on what is left; that is, you convert the quotient x // b to base b. If x // b is 0, there is no rest; x is a single base b digit and that digit is x % b (which also is just x).

In a file named **base_convert.py**, write a recursive function named convert() that will take a non-negative integer num (base 10) and a base b (integer from 2 to 16) and return a string representing the base b number:

```
      def convert(num, b):
      """Recursive function that returns a string representing num in the base b"""

convert(30,4) returns "132"
convert(45,2)) returns "101101"
convert(316,16) returns "13C"
```

Note that for bases > 10, the symbols used for 10, 11, 12, 13, 14, 15 are A, B, C, D, E, F respectively.

Part3: A Teddy Bear Picnic

This question involves a game with teddy bears. The game starts when I give you some bears. You can then repeatedly give back some bears, but you must follow these rules (where n is the number of bears that you currently have):

1. If n is even, then you may give back n/2 bears.
2. If n is divisible by 3 or 4, then you may multiply the last two digits of n and give back this many bears.
3. If n is divisible by 5, then you may give back 42 bears.

The goal of the game is to end up with EXACTLY 42 bears.

For example, suppose that you start with 250 bears. Then you could make these moves:

- Start with 250 bears.
- Since 250 is divisible by 5, you may return 42 of the bears, leaving you with 208 bears.
- Since 208 is even, you may return half of the bears, leaving you with 104 bears.
- Since 104 is even, you may return half of the bears, leaving you with 52 bears.
- Since 52 is divisible by 4, you may multiply the last two digits (resulting in 10) and return these 10 bears. This leaves you with 42 bears.

You have reached the goal!

Write a recursive function to meet this specification:

```
def bears(n):
    """A True return value means that it is possible to win
     the bear game by starting with n bears. A False return value means
     that it is not possible to win the bear game by starting with n
     bears."""
```

```
Examples:
```

- `bears(250) is True (as shown above)`
- `bears(42) is True`
- `bears(53) is False`
- `bears(41) is False`

Although this problem may seem silly at first, it’s an example of a reachability problem, which is a problem that arises in many areas of computer science.https://en.wikipedia.org/wiki/Reachability_problem