Assignment 1

Please read the entire document carefully! There are three parts to this assignment. **Make sure to do all three!**

Contents

1 Permutations in Lexicographic Order

1.1 Specification

We are going to write a Python program to generate all the permutations of the characters in a string. This will give you a chance to review some Python constructs (e.g. strings and lists) and also solidify your understanding of recursion.

Your program ***must*** meet the following specifications. In a file named `perm_lex.py`, You are to write a Python function `perm_gen_lex` that:

- Takes a string as a single input argument. You may assume that the input string will be 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 (i.e. dictionary) order.

  Note: if you follow the pseudocode 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. Adheres to the style in PEP 8.
- Is recursive and follows the pseudocode below.

`perm_gen_lex`(s)

Input: A string s
Output: A list of all permutations of the characters in s in lexicographic order
  for each character c in s **do**
    Form a simpler string, t, by removing c from s
    Generate all permutations of t recursively # *i.e. call* `perm_gen_lex`(t)
    for each permutation p of t **do**
      Add c to the front of p and add the result to your list of permutations
  return the list of permutations

Note, my simple pseudocode never mentions a base case. Your code will need one to function correctly. Think about what it should be and what you should return in that case.

My pseudocode also uses bad variable names (semi-intentionally). I expect better from your code.

1.2 Examples

As an example of what this function does:

```
>>> perm_gen_lex('')
['']
>>> perm_gen_lex('a')
['a']
>>> perm_gen_lex('ab')
['ab', 'ba']
>>> perm_gen_lex('abc')
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> perm_gen_lex('abcd')
['abcd', 'abdc', 'acbd', 'acdb', 'adbc', 'adcb', 'bacd', 'badc', 'bcad',
 'bcda', 'bdac', 'bdca', 'cabd', 'cadb', 'cbad', 'cbda', 'cdab', 'cdba',
 'dabc', 'dacb', 'dbac', 'dbca', 'dcab', 'dcba']
```

Note: For a string with n characters, your program will return a list containing $n!$ strings. This will grow very quickly. If your string contained every lowercase letter a to z (26 letters), the resulting list of permutations would have 403,291,461,126,605,635,584,000,000 $\approx 4.03 \times 10^{26}$ elements. If you want your code to finish in your lifetime, you should probably stick to shorter strings for testing.

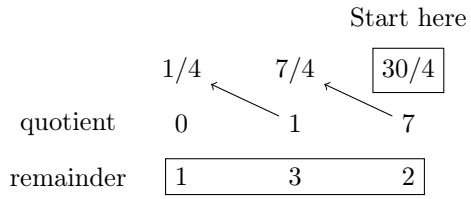
2 Base Conversion

2.1 Specification

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 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 fins the digits for the base b number in reverse order.

Here is an example for converting the base 10 number 30 into base 4:



So, 30 (base 10) ≡ 132 (base 4).

Think about how this is recursive in nature.

- For what values do I know the answer without doing any work?
- How can I turn the problem into a smaller problem?

In a file named `base_convert.py`, write a recursive function named `convert` that will take a non-negative integer in base 10 and a target base (an integer between 2 and 16 inclusive) and returns a string representing the number in the given base.

```
def convert(num: int, base: int) -> str:
    """Returns a string representing num in the given base.

    Implemented recursively.
    """
```

2.2 Examples

As some examples:

```
>>> convert(30, 4)
'132'
>>> convert(45, 2)
'101101'
>>> convert(316, 16)
'13C'
```

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

3 A Teddy Bear Picnic

3.1 Specification

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 together 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 $5 * 2 = 10$) and return these 10 bears. This leaves you with 42 bears.
- 42 bears is the goal! So, we succeeded!

Note that at several of the steps, we had several options for how to proceed; not all of them would have resulted in success.

In a file named `bears.py`, write a recursive function named `bears` to the following specification:

```
def bears(n: int) -> bool:
    """Returns whether it is possible to win the bear game starting with
    n bears.

    Implemented recursively.
    """
```

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.

3.2 Examples

Some examples:

```
>>> bears(250)  # We saw this above
True
>>> bears(42)  # This is true because we're already at the goal.
True
>>> bears(40)
False
>>> bears(53)
False
```

4 Testing

For each part, you are *required* to achieve 100% test coverage. This means that every line of code in your functions *must* be executed at some point in at least one of your tests.

As discussed in the syllabus, when I grade your code, you will ordinarily receive feedback regarding any tests that fail, *unless* you do not have 100% test coverage. In the event you do not have 100% test coverage, the only feedback you will receive is that you need to do more testing. I don't want you using my grading script to do your testing in the last day.

5 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 0, as needed, to remember how to push your local code.