Assignment Guidelines

- This assignment covers material in Module 05.
- Submission details:
  - Solutions to these questions must be placed in files `a05q1.py`, `a05q2.py`, `a05q3.py` and `a05q4.py`.
  - If you submit a solution for the bonus question, put it in `a05bonus.py`. You may copy code from `a05q4.py` into the bonus file.
  - You must be using Python 3 or higher. Do NOT use Python 2.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page.
- Download the testing module from the course web page. Include `import check` in each solution file.
  - When a function returns a floating point value, you must use `check.within` for your testing. Unless told otherwise, you may use a tolerance of `0.00001` in your tests.
- Restrictions:
  - Do not import any modules other than `math` and `check`.
  - Do not use Python constructs from later modules (e.g. loops). Do not use any other Python functions not discussed in class or explicitly allowed elsewhere. See the allowable functions post on Piazza. You are always allowed to define your own helper functions, as long as they meet the assignment restrictions.
  - While you may use global `constants` in your solutions, do not use global `variables` for anything other than testing.
  - Read each question carefully for additional restrictions.

The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
1. **Generating binary strings.**
   Write a recursive function `every_binary_string` that consumes a natural number `n` and returns a list containing every binary string of length `n`. The list must be sorted by the numerical value of the string, e.g., "00" < "01" because 0 < 1.

   For example:
   
   ```
   every_binary_string(0) => ['']
   every_binary_string(1) => ['0', '1']
   every_binary_string(2) => ['00', '01', '10', '11']
   ```

   For full correctness marks, do not sort the list explicitly: do not call any sorting functions, do not swap elements, do not reorder/shuffle any lists that you generate. The list should be **generated** in sorted order.

2. **Counting the digits of a number.**

   For this question you may only use accumulative recursion. Failing to do so will result in an automatic 0 for correctness marks.

   Write a function `count_digits` that consumes a natural number `n` and returns the count of each digit in `n`. More precisely, return a list of length 10, where the `i`\(^{th}\) element of the list contains the number of times that the digit `i` occurs in the base 10 representation of `n`.

   For example:
   
   ```
   count_digits(0) => [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] since the digit 0 occurs once.
   count_digits(440222) => [1, 0, 3, 0, 2, 0, 0, 0, 0, 0] since the digit 0 occurs once, the digit 2 occurs thrice, and the digit 4 occurs twice.
   count_digits(973195) => [0, 1, 0, 1, 0, 1, 0, 1, 0, 2] since digits 1, 3, 5 and 7 occur once, and the digit 9 occurs twice.
   ```

3. **Recursive formula with n terms.**

   Write a function `F` that consumes a natural `n` and returns the sum
   
   \[
   F(n) = F(n-1) + F(n-2) + \ldots + F(2) + F(1) + F(0) + 1
   \]

   for `n>0`; otherwise if `n=0`, the function returns `F(0)=1`.

   Fun fact: you can prove that `F(n) = 2^n`, but do not use this explicit exponential formula in your code. Instead, use a helper function to compute the sum above using structural recursion on the parameter `n`.

   For example, compute these sums explicitly:
   
   ```
   F(0) => 1
   F(1) => F(0)+1 => 1+1 => 2
   F(2) => F(1)+(F(0)+1) => 4
   F(3) => F(2)+(F(1)+(F(0)+1)) => 8
   ```
4. DNA Folding.
You may use any kind of recursion for this question, but do not use loops.

For this problem we will consider an extremely simplified (and inaccurate) model of how a DNA strand can be mutated or “folded” into a new strand. For these purposes, a DNA strand will be represented by a non-empty string of 4 possible characters ("A", "C", "G" or "T") representing its sequence of nucleic acids. For example, "AACTC" is a string representing one possible strand of DNA.

A strand can be folded using two different operations:
- Remove the (F)irst character from the old strand and add it to the end of the new strand.
- Remove the (L)ast character from the old strand and add it to the end of the new strand.

The new strand starts off as the empty string. If we use a sequence of these operations to remove characters from the old strand until there are none left, the new strand will contain all of the old characters, just in a different order.

For example, the strand old="AACTC" can be folded into new="ACTAC":
- Start with: old="AACTC", new=""
- Apply (F): old="ACTC", new="A"
- Apply (L): old="ACT", new="AC"
- Apply (L): old="AC", new="ACT"
- Apply (F): old="C", new="ACTA"
- Apply either (F) or (L): old="", new="ACTAC"

and now old is the empty string, so we are done folding and new="ACTAC".

a. Write a function is_folded that consumes two DNA strands, old and new (strings as described above). The function returns True if it is possible to fold old into new by some sequence of the two operations, False otherwise.

For example:
is_folded("AACTC", "ACTAC") => True as described above.
is_folded("AACTC", "ACATC") => True since "AACTC" can be folded into "ACATC" by alternating between operations (F) and (L).
is_folded("ATGCA", "ACTGA") => False since there is no sequence of the two different operations that folds "ATGCA" into "ACTGA".

You may assume that len(old) = len(new) <= 10. There are multiple ways to solve this problem. Efficiency should not be your primary concern. A relatively inefficient but otherwise correct approach starts by generating every possible strand that can be folded from the old strand, and then checking whether the new strand is one of the generated strands.

b. In the first example above, we applied the sequence of operations F, L, L, F and F to fold old="AACTC" into new="ACTAC". We can encode this sequence of operations as the string "FLLFF". In general, the first character in the string corresponds to the first operation applied to old, the second character corresponds to the second operation applied to old, and so on.

Write a function fold that consumes two parameters: strand, a string for a DNA strand, and e, a string that encodes the sequence of operations as described above. The function returns the DNA strand created by applying the operation sequence e to strand. Note that len(e)
must equal len(strand) to guarantee the returned string contains all of the characters contained by strand.

For example:
fold("AACTC", "FLLFF") => "ACTAC" as described in the previous example.

c. **BONUS**: no hints, advice or debugging help will be provided. The course staff will only answer questions about the wording of the question, e.g., clarifying ambiguities or unspecified behaviour. Design recipe is not required, only function body.

Write a function `find_sequence` that consumes the same parameters as `is_folded`. If there is no sequence of operations that folds `old` into `new`, then `find_sequence` returns `[False, ""]`. Otherwise if a sequence exists, then `find_sequence` returns `[True, seq]` where `seq` is a sequence string encoded as above.

For example:
"FLLFF" and "FLLFL" both fold the DNA sequence "AACTC" to "ACTAC", so `find_sequence("AACTC", "ACATC")` could produce `[True, "FLLFF"]` or `[True, "FLLFL"]"—both are correct output.

**Testing notes**
Testing will not be graded because the design recipe is not required. However, the code will be assessed for correctness, so we recommend that you certify the correctness of your code by testing it thoroughly.

As mentioned in the example above, there can be several correct sequences that fold one DNA strand into another strand. If one sequence exists, then there is at least one other sequence since the choice of the final operation is arbitrary—and it’s always arbitrary! Furthermore, there are \(2^4 = 16\) different sequences that fold `old="CCCC"` to `new="CCCC"`. This makes testing `find_sequence` non-trivial. Please follow the advice below to test your code.

**When we expect `find_sequence` to return `[False, "]"`**, we use:

```
check.expect('find_seq expected to return [False,""]',
            find_sequence("ATGCA", "ACTGA") [0], False)
```

**When we expect `find_sequence` to return a sequence of operations (e.g., `[True, "FLLFL"]`), we don’t test the returned string directly since it is not unique. Instead, we use the `fold` function to test whether the returned string is one of the operation sequences that yields the new strand:**

```
check.expect("find_seq expected to return an operation sequence",
            fold("AACTC", find_sequence("AACTC", "ACATC") [1]),
            "ACATC")
```