Assignment Guidelines.

• This assignment covers material in Module 06.
• Submission details:
  – Solutions to these questions must be placed in files a06q1.py, a06q2.py, a06q3.py, and a06q4.py, respectively, and must be completed using Python 3.
  – 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.
  – Helper functions need design recipe elements but not examples and tests.
• Download the testing module from the course web page. Include import check in each solution file.
• Restrictions:
  – Do not import any modules other than math and check.
  – 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. **Base Conversion.** In assignment 3 you converted a `Str` representing a binary number to a `Nat`. Here we will convert a `Nat` to a `(listof Nat)` representing the binary number.

(a) **Binary Conversion.** Write a function `nat2bin(n)` that consumes a `Nat` and returns a non-empty `(listof (anyof 0 1))`, where items in the list are the binary digits of the `Nat`.

The units digit is the last value in the list, and the most significant digit is the first value in the list.

For example,
- `nat2bin(12)` => `[1, 1, 0, 0]`
- `nat2bin(5)` => `[1, 0, 1]`
- `nat2bin(2)` => `[1, 0]`
- `nat2bin(0)` => `[0]`

Do not use the built-in function `bin` to solve this question.

(b) **Any Base Conversion.** Numbers can be written in bases other than decimal and binary. You may want to review the discussion in assignment 3.

Write a new function `nat2base(n, base)` than consumes two `Nat`. `n` is the number to be converted, and `base` is the base to be converted to. The function returns a non-empty `(listof Nat)` representing `n` in the base given by `base`. (Each value in the returned list will be between 0 and `base-1`, inclusive.)

As before, the units digit is the last value in the list, and the most significant digit is the first value in the list.

You may assume that `base` is at least 2.

For example,
- `nat2base(12, 2)` => `[1, 1, 0, 0]`
- `nat2base(245, 10)` => `[2, 4, 5]`
- `nat2base(326, 5)` => `[2, 3, 0, 1]`
- `nat2base(165, 16)` => `[10, 5]`
- `nat2base(36, 36)` => `[1, 0]`

Hint: solve the problem in base ten first, and compare with your solution from part a. Consider how you need to change your code to work in any base.

2. **Bigger Numbers.** Write a function `find_bigger` that consumes a `(listof Int)`, and returns a `(listof Int)` containing, in order, the values in the list that are bigger than all values that came before in the list.

For example,
- `find_bigger([0, 4, 5, 4])` => `[0, 4, 5]`
- `find_bigger([1, 2, 4, 4])` => `[1, 2, 4]`
- `find_bigger([-2, -4, -4, -1])` => `[-2, -1]`
- `find_bigger([])` => `[]`

The first item in the consumed list is always included in the returned list.

Do not mutate the list your function consumes.
3. **Palindromes.** A palindrome is a string that is the same forwards and backwards. For example, 'racecar', 'anna', and 'albophobia' (the fear of long palindromes) are all palindromes, but 'abc' and 'sata' are not. You may assume each Str contains only lowercase letters.

(a) **Checking.** The following recursive function is taken from the CS 116 course notes, Module 05. Rewrite it using a **while** loop, without using recursion.

```python
# is_palindrome(s) Return True if s is a palindrome.
# is_palindrome: Str -> Bool
# Examples:
# is_palindrome ('racecar') => True
# is_palindrome ('sata') => False

def is_palindrome(s):
    if len(s) < 2:
        return True
    else:
        return s[0] == s[-1] and is_palindrome(s[1:-1])
```

(b) **Find Longest.** We say that a string is a **subpalindrome of s** it is both a substring of s and a palindrome. Complete the function longest_subpalindrome, that consumes a Str that contains only lowercase letters, and returns the longest subpalindrome. If there are multiple subpalindromes that are the same size, then return the one that comes nearest the beginning of the Str. For example,

```python
longest_subpalindrome('acaba') => 'aca'
longest_subpalindrome('anna') => 'anna'
longest_subpalindrome('sata') => 'ata'
longest_subpalindrome('cba') => 'c'
longest_subpalindrome('aracecarwinsarace') => 'racecar'
```

4. **Mountain Ranges (Dyck Paths).** In this question we will be concerned with drawing mountain ranges. These diagrams are used to study **Dyck paths**, an interesting combinatorial object.

The input to your functions will be desc, which is a Str that describes the mountain range. desc is guaranteed to have the following properties:

1. it contains only the characters '+' and '-'.
2. The two characters appear the same number of times.
3. Every substring of desc that starts at zero has at least as many '+' characters as '-'.

The output starts at the bottom left. In each step, it looks at the next character in desc, and moves to the right, and either up (for '+') or down (for '-'). It writes a slash character / when in moves up, and a backslash character \ when in moves down.

For example, if testdata = '+++++-----------------------', the diagram for testdata will look like:

```
/\ \\
/ / \\
/ / \\
/ / \\
/ / \\
/ \\
```
(a) Maximum height. First write a function `max_height(desc)` that returns the maximum height reached. This number will be the number of lines required to store the mountain range diagram. For example,

```
max_height(testdata) => 4
max_height('+-') => 1
max_height('+++---') => 3
max_height('+-+-+-') => 1
max_height('') => 0
```

(b) Mountain Rendering. Write a function `render_mountain(desc)`.

Your function should return a `(listof Str)`, where the first line is the top line of the image of the mountain range, and the last line is the bottom line of the image. All lines are of the same length, and are filled with spaces.

*Hint: since you cannot modify a Str, it may help to store your data as a `(listof (listof Str))`, where each Str is of length 1.*

There are some display issues you may encounter:

1. Since the function returns a `(listof Str)` it does not display nicely on the screen. Also, each backslash will appear twice. For example, once your `render_mountain` function is complete:

```
render_mountain('+++---')
=> [' /\ ', ' /\/ \ ', '/\ ', '/\ ']
```

To get nice output, you are encouraged to use the following or similar code to view the output:

```
for line in render_mountain('+++---'):
    print(line)
```

This should print:

```
/\ \\
/\/ \ \\
/ \\
```

(You are not required to test the appearance. Test the exact value returned by the function.)

2. Python uses the backslash character `\` to form *escape sequences*. The only one you have seen is the newline, `'\n'`.

But there are others, including `'\'` and `'"'`, each of which is a Str containing only a quotation mark. So if you attempt to end a Str with a backslash, like `'\'`, Python will interpret this as a quotation mark *as part of the Str*, not ending the Str.

As a result, at the end of the last line in your tests, you must type an extra backslash, and the output from your function may not appear as you might expect.

```
check.expect('simple',
    render_mountain('+++---'),
    [
        ' /\ ',
        ' /\/ \ ',
        '/\ ', # <-- extra backslash is required!
    ])
```

You *may* escape all the backslashes, but it is not required or encouraged.

You may like to use the following constants: `up = '/'` and `down = '\\'`.

Despite appearances, each of these Str is of length 1.