Assignment Guidelines.

- This assignment covers material in Modules 6 and 7.
- Submission details:
  - Solutions to these questions must be placed in files `a07q1.rkt`, `a07q2.rkt`, `a07q3.rkt`, and `a07q4.rkt`, respectively, and must be completed using Racket *Intermediate Student with lambda*.
  - Unless otherwise indicated in the question you may use only the built-in functions and special forms introduced in the lecture slides from CS115 up to and including the modules covered by this assignment.
  - 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 CS115 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page.
  - For the design recipe, helper functions only require a purpose, a contract and an example.
- Restrictions:
  - You should expect to use recursion on every question.
  - Read each question carefully for additional restrictions.

    ! Do not use `map`, `foldr`, `filter`, `length`, `append`, or `range` on this assignment.

- **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. **Faro Shuffle.** A faro shuffle involves splitting a deck of cards into two piles, then taking cards exactly alternating between the two piles.

Exercise

Write a function `(faro-shuffle L1 L2)`, where each parameter is a `(listof Any)`. The function should return a list which contains all the items in `L1` and `L2`, interleaved. That is, the first item in the result is from `L1`, the second is from `L2`, then it alternates.
If `L1` and `L2` are not of equal length, the extra items should be included at the end.

```
(check-expect (faro-shuffle (list "a" "b" "c") (list "A" "B" "C"))
              (list "a" "A" "b" "B" "c" "C"))
(check-expect (faro-shuffle (list 1 3 5) (list 2 4 6 8 10))
              (list 1 2 3 4 5 6 8 10))
```

2. **Run Length Encoding.** If a list contains "runs" of repeated values, it is possible to store the information in a more compact manner using a *run-length encoding*. For example, $[4,4,4,4,4,4,4,6,6,6,3,3,3,3,3]$ contains seven copies of 4, three copies of 6, then five copies of 3.

This technique was used by some early image compression algorithms before GIF (which is itself rather obsolete), but is still used in fax machines. (Has anybody ever seen a fax machine?)

```
;; A Run is a (list Nat Any), where
;;   * the first item counts the number of items
;;   * the second item indicates the item.
```

For example, `(list 3 3 3 3 3)` would be represented as the `Run` `(list 5 3)`, and `(list "foo" "foo")` would be represented as `(list 2 "foo")`.

Exercise

Write a function `(encode-rle L)`. the function consumes a `(listof Any)` and returns the shortest-possible `(listof Run)` representing the same pattern.
For example,

```
(check-expect (encode-rle (list "hee")) (list (list 1 "hee")))
(check-expect (encode-rle (list "hee" "hee" "hee" "ho" "ho" "hee"))
              (list (list 3 "hee") (list 2 "ho") (list 1 "hee")))
(check-expect (encode-rle (list 4 4 4 4 4 4 6 6 6 3 3 3 3 3))
              (list (list 7 4) (list 3 6) (list 5 3)))
```

Additional tests are in the interface file.

Hint

You may want to write functions to:

- *count* the duplicates at the front of a list, and
- *remove* the duplicates from the front of a list.

3. **Counting.**

Exercise

Write a function `(count-targets targets L)`. It consumes two `(listof Any)`, and returns a `(listof Nat)` that indicates how many times each item in `targets` appears in `L`. For example,

```
(check-expect (count-targets (list 1 2 3) (list 1 2 1 2 1 1 1 7))
              (list 5 2 0))
(check-expect (count-targets (list "a" "b" "b" "q")
                             (list "a" "c" "b" "b" "q" "a" "a" "z"))
              (list 3 2 2 1))
```

4. Big Numbers. This question explores how we can represent arbitrarily big numbers, even if the native numbers have a limited range. Instead of using `Nat`, we with use the following data definition:

```
;; A Digit is a Nat
;; Requires: the value is less than 10.
```

To represent multi-digit numbers, we will use a list of digits:

```
;; a DigitList is a (listof Digit)
;; Requires: the last value in the list is not zero.

;; if you prefer:

;; a DigitList is either:
;;   '() or
;;   (cons d L) where d is a Digit and L is a digitList.
;; Requires: the last value in the list is not zero.
```

This uses a list to represent a number in base 10. The digits are stored smallest first, so they seem backwards. For example, the number 245 is represented `(list 5 4 2)`. Also note: zero is represented as the empty list, `'()`.

(Note that we do not ordinarily write numbers with leading zeros; we write "42" not "042". That is the meaning of the `Requires`.)

Reducing a number by 1 is called *decrementing*. Here you will write a function to decrement a number represented as a `DigitList`.

Exercise

Write a function `(decrement digits)` that consumes a `DigitList`, and returns the `DigitList` representing the number one smaller.

```
(check-expect (decrement (list 5 3 2)) (list 4 3 2))       ;   235 - 1 = 234
(check-expect (decrement (list 0 2)) (list 9 1))           ;    20 - 1 = 19
(check-expect (decrement (list 0 1)) (list 9))             ;    10 - 1 = 9
(check-expect (decrement (list 0 0 5)) (list 9 9 4))       ;   500 - 1 = 499
(check-expect (decrement (list 0 0 0 0 1)) (list 9 9 9 9)) ; 10000 - 1 = 9999
```

Hint

Carefully think about how you do subtraction.
What does it mean to "borrow"? If you are subtracting 1, when do you need to borrow?