Assignment Guidelines.

- This assignment covers material in Modules 6 and 7.
- Submission details:
  - Solutions to these questions must be placed in files `a06q1.rkt`, `a06q2.rkt`, `a06q3.rkt`, and `a06q4.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. Counting up and down.

Exercise

Write a recursive function `(my-range start end step)` that duplicates the behaviour of the built-in **`range`** function. For example,

```
(check-expect (my-range 4 13 3) (list 4 7 10))
(check-expect (my-range 17 8 -2) (list 17 15 13 11 9))
(check-expect (my-range 1.7 14.5 3.2) (list 1.7 4.9 8.1 11.3))
```

Hint

Probably you want to `Require` that `step` is not zero.
You might need to treat separately if `step` is positive or negative.

2. Extracting a value from a list.

The built-in function `(list-ref L i)` consumes a `(listof Any)` and a `Nat`, and returns the `i`th item in `L`, where `(first L)` is item zero.

Exercise

Using recursion, write a function `(my-list-ref L i)` that duplicates the behaviour of `list-ref`.

```
(check-expect (my-list-ref (list "a" "b" "c") 0) "a")
(check-expect (my-list-ref (list "a" "b" "c") 2) "c")
```

3. Searching in Strings.

Exercise

Write a function `(find s pattern)` that looks through `s` and returns the smallest index in `s` where `pattern` exists as a substring. If `pattern` does not appear in `s`, return `-1`.
For example,

```
(check-expect (find "one fish two fish red fish blue fish" "fish") 4)
(check-expect (find "one fish two fish red fish blue fish" "crab") -1)
(check-expect (find "one fish two fish red fish blue fish" "red fish") 18)
(check-expect (find "abracadabra" "ab") 0)
(check-expect (find "haystack" "a needle") -1)
(check-expect (find "" "anything") -1)
```

Hint

Write a helper function that simply determines if `s` contains `pattern`.

4. Skipping.

Exercise

Write a function `(skipping L n)` that consumes a `(listof Any)` and a `Nat`. It returns a `(listof Any)` that contains the first item in `L`, and after any item, it skips the next `n` items.
For example,

```
(check-expect (skipping (list 2 3 5 7 11 13 17) 0) (list 2 3 5 7 11 13 17))
(check-expect (skipping (list 2 3 5 7 11 13 17) 1) (list 2 5 11 17))
(check-expect (skipping (list 2 3 5 7 11 13 17) 2) (list 2 7 17))
(check-expect (skipping (list 2 3 5 7 11 13 17) 3) (list 2 11))
```

Like so:
Skip 1: `(list 2 ~~3~~ 5 ~~7~~ 11 ~~13~~ 17) => (list 2 5 11 17)`
Skip 2: `(list 2 ~~3~~ ~~5~~ 7 ~~11~~ ~~13~~ 17) => (list 2 7 17)`
Skip 3: `(list 2 ~~3~~ ~~5~~ ~~7~~ 11 ~~13~~ ~~17~~) => (list 2 11)`