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. **Selection Sort.** One way to sort a list of values is to find the smallest item in the list, sort all the items other than this smallest value, and then put the smallest value in front of the (now sorted) list of other values. This is called *selection sort*.

   For example, to sort \[4, 6, 8, 3, 5\]:
   - “Select” the smallest item in the list, which is 3;
   - Find the shorter list that omits this item, that is, \[4, 6, 8, 5\].
   - Using recursion, sort this shorter list, giving \[4, 5, 6, 8\].
   - Put the smallest item found earlier before this list, giving \[3, 4, 5, 6, 8\].

   **Exercise**
   Write a function `smallest L` that consumes a non-empty `(listof Num)` and returns the smallest value in L.
   For example,
   `smallest (list 32 8 6 -3 -7 1 42))` => `-7`

   **Exercise**
   Write a function `remove-item L item` that consumes a `(listof Any)` and an Any. It returns a list containing all the contents of L except item.
   If `item` appears more than once in L, remove only the first copy.
   It is not an error if `item` does not appear in L.

   You can now identify the smallest value in a list (using `smallest`) and remove that smallest item from a list (using `remove-item`).

   Recursion on a list does not necessarily need to work only by splitting into first and rest. As long as the list is getting shorter in some way, the recursion will finish.

   **Exercise**
   Write a function `selsort L` that consumes a `(listof Num)`. The function uses selection sort to sort L into increasing order.
   For example,
   `selsort (list 2 4 6 0 1))` => `(list 0 1 2 4 6)`

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, \[4, 4, 4, 4, 4, 4, 6, 6, 6, 3, 3, 3, 3, 3\] 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.

   ```
   ;; 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 `decode-rle L`. The function consumes a `(listof Run)` and returns the `(listof Any)` representing the same pattern.
   For example,
   `decode-rle (list (list 3 "hee") (list 2 "ho")))` => `(list "hee" "hee" "hee" "ho" "ho")`

   You *may* use `append` to help solve this question. It is possible to solve it without using `append`. 


3. **Items from a list.** Often we want to extract a few values from a list, by knowing their index in the list. For a single value, there is a built-in function, `list-ref`, to do this, and we wrote our own recursive replacement on the previous assignment.

If `itemL` is a `(listof Nat)` containing the indices of multiple values to retrieve, I could call `my-list-ref` repeatedly, recursing on `itemL`:

```scheme
;; extract items one by one:
(define (extract-items L itemL)
  (cond [(empty? itemL) '()]
        [else
         (cons (my-list-ref L (first itemL))
               (extract-items L (rest itemL))))])
```

Each time this function calls `my-list-ref`, the computer needs to walk through the list from the beginning to the location of the desired item. Given the following, it will run `my-list-ref` many times:

```scheme
(extract-items (range 0 100 1) (list 99 99 99 99 99 99 99 99 99 99)) ; big test
```

This is *inefficient*; it ends up walking through all of `L` many times, doing the same work each time. If `itemL` is sorted, we can solve the problem more efficiently, and walk through `L` only once.

---

**Exercise**

Write `(extract-items L itemL)`. It consumes a `(listof Any)`, and a `(listof Nat)` where the values are non-decreasing.

The function should recurse through `L` only once, and return the `(listof Any)` containing the items at the locations specified in `itemL`.

For example,

```scheme
(extract-items (list "a" "b" "c" "d" "e") (list 0 2 2 3)) => (list "a" "c" "c" "d")
```

---

**Hint**

Add a counter to keep track of how far through the list you are so far.

---

**!** Do not use `list-ref` or `my-list-ref` in your solution.

You must recurse through `L` only once.
4. Adding 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 will use the following data definition:

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

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

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

;; if you prefer:

```scheme
;; 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, `'(0).

(a) Carry.

Write a function `(one-more-than digits)` that consumes a `DigitList`, and returns the `DigitList` representing the number one greater.

`(one-more-than (list 5 4 2)) => (list 6 4 2)`
`(one-more-than (list 9 1)) => (list 0 2)`
`(one-more-than (list 9 9 9)) => (list 0 0 0 1)`

(b) Recursive addition. Adding two numbers can be viewed as a recursive process.

First consider small numbers with no carry. $5321 + 57 = 5378$. This can be viewed as $532 + 5 = 537$ followed by $1 + 7 = 8$. Recursively compute $532 + 5$, giving 537; so the final answer is 537 followed by 8, i.e. 5378.

Now consider an addition with carry. $5346 + 57$. Considering the units digits, $6 + 7 = 13$. This has a carry; the units digits of the sum will be 3, but the rest of the work is `one more than` $534 + 5$; one more than 539 is 540, so answer is 540 followed by 3, so 5403.

Write a function `(add D1 D2)` that consumes two `DigitList`, and returns the `DigitList` representing their sum.

```scheme
(check-expect (add (list) (list 4)) (list 4)) ; 0 + 4 = 4
(check-expect (add (list 3) (list 4)) (list 7)) ; 3 + 4 = 7
(check-expect (add (list 7) (list 6)) (list 3 1)) ; 7 + 6 = 13
(check-expect (add (list 5 4 2) (list 7 6 5)) (list 2 1 8)) ; 245 + 567 = 812
```

Do not convert your `DigitList` to a regular `Nat`. Do the computation using the recursive structure of the `DigitList.`