• Unless otherwise indicated by the question, you may only use the built-in functions and special forms introduced in the lecture slides from CS115 up to and including the modules covered by this assignment. A list of functions described in each module of the lecture slides can be found on LEARN.
• For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch, including helper functions.
• 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.
• Do not send any code files by email to your instructors or tutors. Course staff will not accept it as an assignment submission. Course staff will not debug code emailed to them.
• Read each question carefully for restrictions.
• Test data for all questions will always meet the stated assumptions for consumed values.
• Download the interface file from the course Web page to ensure that all function names are spelled correctly, and that each function has the correct number and order of parameters.
• Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include references to the parameter names of your functions.
• Check MarkUs and your basic test results to ensure that your files were properly submitted. In most cases, solutions that do not pass the basic tests will not receive any correctness marks.
• Any string constant values must exactly match the descriptions in the questions. Any discrepancies in your solutions may lead to a severe loss of correctness marks. Basic tests results will catch many, but not necessarily all of these types of errors.
• Read the course Web page for more information on assignment policies and how to organize and submit your work. Follow the instructions in the Style Guide. Your solutions should be placed in files a6qY.rkt, where Y is a value from 1 to 4.
• Since each file you submit will contain more than one function, **it is very important that your code runs.** If your code does not run then none of the functions can be tested for correctness.
• **Plagiarism:** The following applies to all assignments in CS115.
  – Be sure to read the Plagiarism section at: [https://www.student.cs.uwaterloo.ca/~cs115/#assignments](https://www.student.cs.uwaterloo.ca/~cs115/#assignments)

**Language level:** Beginning Student with List Abbreviations

**Coverage:** Module 6

This assignment uses the following structure definitions.

```scheme
;; A Step is a (list up down)
;; requires: up, down are both Nat
;;          up <= down

(define-struct course
  (subject number title))
;; A Course is a
```
(define-struct student
     (ID name age courses))
;; A Student is a
;; (make-student Nat Str Nat (listof Course))

The examples in this assignment use the following constant definitions.

(define course1
  (make-course "CS" 115 "An Introduction to Computer Science 1"))
(define course2
  (make-course "CS" 116 "An Introduction to Computer Science 2"))
(define course3
  (make-course "MATH" 135 "Algebra for Honours Math"))
(define course4
  (make-course "MATH" 136 "Linear Algebra 1 for Honours Math"))
(define stud1 (make-student 20493453 "Alice" 19
  (list course1 course3)))
(define stud2 (make-student 20563432 "Bob" 20
  (list course2 course3)))
(define stud3 (make-student 20544632 "Carl" 20
  (list course2)))
(define stud4 (make-student 20565453 "David" 21
  (list course1 course4)))
(define stud-list (list stud1 stud2 stud3 stud4))

1. Write the function do-they-meet which consumes four natural numbers lower, upper, inc and dec, and produces a (listof Step) or false. Specifically, the function should
   • determine whether it is possible to count up from lower by an amount inc and arrive at the same value by counting down from upper by an amount dec in the same number of steps (including possibly zero steps);
   • if it is not possible, then the function should produce false, and
   • if it is possible, then the function should produce a list of Step displaying each step of the computation, starting with (list lower upper) and continuing until the up and down fields of the final Step are equal.

For example:
   • (do-they-meet 3 9 1 2)
     => (list (list 3 9) (list 4 7) (list 5 5))
     (in this example we increment the up value by 1, and we decrease the down value by 2)
   • (do-they-meet 17 17 8 3) => (list (list 17 17))
   • (do-they-meet 20 30 4 5) => false
     (the up values would be 20, 24, 28, 32... etc., the down values would be 30, 25, 20, 15...,
etc. Note that they do not match using the same number of steps)

- (do-they-meet 20 30 1 1)
  => (list (list 20 30) (list 21 29) (list 22 28) (list 23 27)
       (list 24 26) (list 25 25))
- (do-they-meet 5 14 0 3)
  => (list (list 5 14) (list 5 11) (list 5 8) (list 5 5))

2. A **perfect number** is a positive integer that is equal to the sum of its positive divisors excluding itself. For example, 28 is a perfect number since it has divisors 1, 2, 4, 7 and 14 and 

\[ 28 = 1 + 2 + 4 + 7 + 14. \]

Write a predicate `perfect?` which consumes a non–zero natural number \( n \) and produces `true` if \( n \) is a perfect number, and `false` otherwise. For example,

- `(perfect? 5)` => `false`, and
- `(perfect? 28)` => `true`.

Note that you are **not** allowed to use a function that searches through a list of known perfect numbers.

3. Modify the `insert` and `sort` functions from Module 06, Slides 29-36, into `posn-insert` and `posn-sort`, to sort a list of `Posn` structures `(alop)` according to the following rules:

- Sort the list in increasing order by distance from the origin (i.e. from `(make-posn 0 0)`).
- In the case of two (or more) `Posn` values being the same distance from the origin, sort by increasing order of \( x \)-coordinates.
- In the case of two (or more) `Posn` values being the same distance from the origin and having the same \( x \)-coordinate, sort by increasing order of \( y \)-coordinates.

For example:

- `posn-sort (list (make-posn 3 4)
  (make-posn 0 1)
  (make-posn 1 0)
  (make-posn -3 4)
  (make-posn -3 -4))`

  => `(list (make-posn 0 1)
           (make-posn 1 0)
           (make-posn -3 -4)
           (make-posn -3 4)
           (make-posn 3 4))`

Recall that the distance from the origin for a `Posn`, \( p \), is calculated as:

\[
\sqrt{(\text{posn-x } p)^2 + (\text{posn-y } p)^2}
\]

Note that for this question, you may work with the square of the distance, rather than the distance itself, to avoid processing inexact values. The sort order will not be changed if you consistently use the square of the distance instead of the distance itself. However, note the `Posn` values will always be exact, so you can still use `check-expect` when testing these functions, even if you do use the actual distance.
4. Complete a Racket function called `oldest-student` that consumes a non-empty list of Students (anelos) and produces the name of the oldest student in anelos. If there is a tie with respect to ages, then produce the student that occurs first in the list. For example:
   - `(oldest-student (list stud1 stud3)) => "Carl"
   - `(oldest-student (list stud2 stud3)) => "Bob"
   - `(oldest-student stud-list) => "David"