Homework 3: Racket Lists and Java Arrays

In this homework, we will explore our theme from class this week: list data structures in Racket and arrays in Java. The bulk of the assignment focuses on implementing lists in Racket, and we'll also continue to emphasize the importance of testing and style.

> **Important**
> For this entire assignment, your solutions should use **recursion**. If you have looked ahead to future weeks, higher-order functions such as **`map`**, **`filter`**, **`sort`**, and **`foldr`** are expressly forbidden! You will practice using higher-order functions in the next assignment.

Starter code is on the CS60 GitHub repository. See the README in the repository about how to clone the code to your local machine.

You are welcome (and encouraged) to buddy or pair program again for all parts of this assignment! (The syllabus can remind you of the differences.) If you have not tried buddy or pair programming, give it a try! Feel free to use the "Search for Teammates" post on Piazza to identify potential programming buddies, or ask the teaching team to help you find a partner during lab hours.

We are really excited to help people on Piazza - so please ask questions early and often! Some more tips:

- For code help and questions, please use Piazza with the **#hw3** tag.
- **Your code will be graded anonymously.** Please do not include your name anywhere in your submission.
- **If you choose to pair program, remember to add your partner to each Gradescope submission.**

- Your profs & the awesome CS60 Grutoring team

Problem 1: remove-all

- Learning Goal: Write recursive functions in Racket (with lists)
- Prerequisites: Racket syntax for lists & testing in Racket
- Starter file: **`remove-all.rkt`** and **`remove-all_tests.rkt`**
- Submit: **`remove-all.rkt`** and **`remove-all_tests.rkt`** on the Gradescope

Examine the starter files, which will help you out below by collecting all the function signatures and the *provided* test cases already in place.

In Racket, the keyword **`remove`** can be used to create a list in which the first top-level instance of a value is removed from an existing list. The syntax is **`(remove elem L)`**, where **`elem`** is the value to be removed and **`L`** is the list from which to remove it, like so:

```
None
> (define L '(1 2 3 4 2))
> (remove 2 L)
'(1 3 4 2)
```

Your task is to implement and test the **`remove-all`** function, which is identical to **`remove`** except that **all** instances of **`elem`** have been removed. For example, we could test **`remove`** with the test:

```
None
(check-equal? (remove-all 0 '(1 0 1 0 1 0))  '(1 1 1))
```

For this problem, you only need to remove *top-level* instances of **`elem`**. By top-level, we mean that you do not have to worry about (nor should you remove) instances that appear in sublists of **`list`**:

```
None
> (remove-all 99 '(1 2 99 (1 2 99) 1 2 99))
'(1 2 (1 2 99) 1 2)
```

Write test cases

Write at least two simple but useful test cases for **`remove-all`**.

A few reminders from the previous homework:

- We have provided complex test cases for each function.
    - However, they are not very helpful for debugging your code! This is because complex test cases can only tell you *if* your code works or does not work. They do not provide insight into *the ways* in which code does and does not work.
    - You will be graded on the extent to which your "simple" test cases are simple. For example, if we did not provide a test case that checks *only* the base case of the function, you should do that. If there are two possible base cases, you should have one simple test case for each of these. In general, if there are different "paths" for your code depending on the input, you should test each of those inputs.
    - Writing these test cases can be really helpful for thinking through the behavior of the code, and we want you to start seeing how good test cases help you debug! If you are unsure of whether your test cases are "good enough", feel free to talk with course staff.
- To make it easy for us to identify your additional test cases, you are required to mark provided and student test cases with `; provided tests` and `; student tests`, respectively.

New from last week!

- Sometimes, you may need to add complex test cases in addition to our provided test cases. For example, if you find that your code passes your simple tests but does not pass one of our provided tests, you might try adding a new test that removes some complexity from a provided test.
- Sometimes, you may need to write more complex test cases than even our provided test cases. Have you identified a tricky case? Add it in.

Write `remove-all`

Remember: Use recursion! You should not use `remove` in your solution.

Problem 2: **`prefix?`** and **`sublist?`**

- Learning Goal: Write recursive functions in Racket (with lists)
- Prerequisites: Racket syntax for lists & testing in Racket
- Submit: **`prefix_sublist.rkt`** and **`prefix_sublist_tests.rkt`** on the Gradescope

`prefix?` takes in two lists **`P`** and **`L`**. The function returns **`#t`** if and only if the list **`P`** is identical to the initial elements of the list **`L`**, i.e if **`P`** is the prefix of **`L`**. Otherwise, it returns **`#f`**. Consider the empty list as a prefix of *any* list **`L`** (even itself!).

Function signature:

```
None
(define (prefix? P L)
```

Provided test cases:

```
None
(check-true  (prefix? '()    '(s p a m)))
(check-true  (prefix? '(s p) '(s p a m)))
(check-false (prefix? '(s m) '(s p a m)))
(check-false (prefix? '(p a) '(s p a m)))
```

`sublist?` takes in two lists **`S`** and **`L`**. The function returns **`#t`** if and only if the list **`S`** is identical to *some* set of *consecutive* elements of the list **`L`**, i.e. **`S`** is a sublist of **`L`**. Otherwise, it returns **`#f`**. Consider the empty list to be a sublist of *any* list. However, a *non-empty* list is not a sublist of the empty list.

Function signature:

```
None
(define (sublist? S L)
```

Provided test cases:

```
None
(check-true  (sublist? '()    '(s p a m)))
(check-true  (sublist? '(s p) '(s p a m)))
(check-false (sublist? '(s m) '(s p a m)))
(check-true  (sublist? '(p a) '(s p a m)))
```

Tasks

Prepare your solution files

- Create a Racket file **`prefix_sublist.rkt`**
  - Double-check that the file says the correct language (i.e. Racket):
    `#lang racket`
  - Add code to tell the autograder what functions you will write:
    `(provide prefix? sublist?)`
  - Write function comments and stubs for **`prefix?`** and **`sublist?`**
- Create a Racket file **`prefix_sublist_tests.rkt`**
  - Double-check that the file says the correct language (i.e. Racket):
    `#lang racket`
  - Add code that allows you to use the testing framework:
    `(require rackunit)`
  - Add code that allows you to use your (future) function from **`prefix_sublist.rkt`**:
    `(require "prefix_sublist.rkt")`

Write function comments and stubs, and test cases

- Write function comments and stubs for **`prefix?`** and **`sublist?`**
- Write at least two useful test cases for **`prefix?`** and **`sublist?`**. Mark your tests with the comment **`; student tests`**. Be sure to test simple cases not tested in the provided tests. Then add tests as needed to test any tricky cases that you can think of.
- Copy-and-paste the tests provided above into your file. Mark these tests with the comment **`; provided tests`**

Write **`prefix?`** and **`sublist?`**

- Use recursion!
- To implement **`sublist?`**, use **`prefix?`** to do some of the work!

Problem 3: `enumerate`

- Learning Goal: Write recursive functions in Racket (with lists)
- Prerequisites: Racket syntax for lists & testing in Racket
- Submit: **`enumerate.rkt`** and **`enumerate_tests.rkt`** on the Gradescope

`enumerate` takes in a list **`L`**. The function returns a list identical to **`L`**, except that each element from the original input is now a list with two elements **`(index elem)`**, where **`index`** is the 0-indexed index of the element and **`elem`** is the original element. (You may find the provided test cases useful to help clarify functionality.)

Function signature:

```
None
(define (enumerate L)
```

Provided test cases:

```
None
(check-equal? (enumerate '(jan feb mar apr))
 '((0 jan) (1 feb) (2 mar) (3 apr)))

(check-equal? (enumerate '(0 I II III IV V VI))
                         '((0 0) (1 I) (2 II) (3 III) (4 IV) (5 V) (6
VI)))

(check-equal? (enumerate '())  '())
```

Tasks

Prepare your solution files, write function comments and stubs, and write simple test cases

- Use the same instructions as Problem 2

Write **`enumerate`**

- We suggest that you create a recursive helper function with *more* arguments than **`enumerate`**. Then, have **`enumerate`** call the helper function with a reasonable default value. While you do not have to write test cases for your helper function, you do have to write a function comment.

Optional Aside:

- This function creates *association lists*, or "a-lists" for short. An association list is a list of lists, and it gives us a way to store values that can be looked up using a *key* (like a **`dict`** in Python!). In **`enumerate`**,

the first element of each sublist is the key, and the rest of each sublist is the corresponding value. We will use association lists later in this assignment to score Scrabble.

- You can read more about association lists in Racket (special shout-out to whoever wrote the adorable Harry Potter example). If you are interested, check out **`enumerate`** in Python, which does the same thing!!

```
list(enumerate(['jan', 'feb', 'mar', 'apr']))
```

Problem 4: `superreverse` and `duperreverse`

- Learning Goal: Write recursive functions in Racket (with lists)
- Prerequisites: Racket syntax for lists & testing in Racket
- Submit: **`superduper.rkt`** and **`superduper_tests.rkt`** on the Gradescope

Recall that `(reverse '(1 2 3))` ⇒ `'(3 2 1)`. **`superreverse`** is a variant of **`reverse`** with signature `(define (superreverse L)`. **`L`** is a list that contains zero or more lists (and only lists) as elements. The output is a list identical to **`L`**, except that all top-level lists are reversed. Note: No deeper structures should be altered.

Here are two examples of **`superreverse`** in action. In the second, you will notice notation like **`#\a`**. You do not have to worry about what this notation means; in brief, it specifies the variable is a single character.[1]

```
None
(check-equal? (superreverse '( (1 2 3) (4 5 6 (7 8) 9 ) ))
                            '( (3 2 1) (9 (7 8) 6 5 4) ) )

(check-equal? (superreverse '( (1 2 3) (4 5 6) (#\k #\o #\o #\l) (#\a #\m) ))
                            '( (3 2 1) (6 5 4) (#\l #\o #\o #\k) (#\m #\a) ) )
```

`duperreverse` is a variant of **`superreverse`** with signature `(define (duperreverse L)`. **`L`** is a list in which each element may also be a list. The output is a **deep** reversal of **`L`** that reverses all lists, not just the top-level ones.

Provided tests:

```
None
(check-equal? (duperreverse '( (1 2 3) (4 5 6) 42 ("k" "o" "o" "l") ("a" "m") ))
                            '( ("m" "a") ("l" "o" "o" "k") 42 (6 5 4) (3 2 1) ))

(check-equal? (duperreverse '( (1 2 3) (4 5 6 (7 8) 9) ))
                            '( (9 (8 7) 6 5 4) (3 2 1) ))
```

Tasks

Prepare your solution files, write function comments and stubs, and write simple test cases

- Rinse and repeat from the previous problems. Make sure to spell the function names correctly.
- You are doing great! Keep it up!

Write **`superreverse`**

[1] For the curious, this is Racket's way of representing single-character literals. You can use this *single-character data type* to emphasize when a variable differs from both symbols (**`'symbol`**) and strings (**`"string"`**).

- You may (should!) use the built-in **`reverse`** function in your implementation of **`superreverse`**.

Write **`duperreverse`**

- Notice that members of **`L`** are not necessarily lists, so you should not use **`superreverse`**.
- You can use the Racket predicate **`(list? x)`** to determine whether or not a value is a list.
- Work your way from the front of the list. You will not need to use **`last`**.

Problem 5: `subbag?` and `Scrabble`

- Learning Goal: Write recursive functions in Racket (with lists)
- Prerequisites: Racket syntax, testing in Racket, and Racket lists
- Submit: **`scrabble.rkt`** and **`scrabble_tests.rkt`** on the Gradescope

Note that the first part of this problem (`subbag?`) is required for a passing grade on this portion of the assignment. The second part of this problem (Scrabble `best-word`) is more challenging and will allow you to earn full points (an A).

Setup for `subbag?`

A *bag* is a mathematical term for a set in which there may be more than one instance of the same element. Write the Racket function that begins `(define (subbag? S B)`, where inputs **`S`** and **`B`** are both lists. This function evaluates to **`#t`** if and only if the list **`B`** contains *all* of the elements in **`S`** with a count *at least as large*. In contrast to **`sublist?`**, the order of the elements makes no difference. For example, let us say **`S`** contains three 42's. Then, for **`S`** to be a subbag of **`B`**, **`B`** would also need to contain at least three 42's.

Here are a couple of tests that help illustrate **`subbag?`**:

```
None
(check-true  (subbag? '()      '(s p a m s)) )
(check-true  (subbag? '(s s)   '(s p a m s)) )
(check-true  (subbag? '(s m)   '(s p a m s)) )
(check-true  (subbag? '(a p)   '(s p a m s)) )
(check-false (subbag? '(a m a) '(s p a m s)) )
(check-true  (subbag? '(a s)   '(s a))       )
```

Test and implement `subbag?`

Prepare your solution files

- For simplicity, go ahead and set up the function comments, stubs, and tests only for **`subbag?`**; that is, for now, ignore the rest of the functions for this problem.

Write **`subbag?`**

- In addition to the standard list functions **`empty?`**, **`first`**, and **`rest`**, we used the built-in list functions **`remove`** and **`member`**.

If you decide to stop here, you still need to provide a function stub for **`best-word`** so that the autograder does not error out. Read enough of the next section to understand the input and output of the **`best-word`** procedure. Then define the **`best-word`** procedure using the signature provided, and have the procedure return a meaningless output ***of the specified type***. Then **submit on Gradescope** to confirm the autograder is

working, even if you plan to work on Scrabble more later. If you do everything correctly, your submission should pass the **`subbag?`** test cases but not the **`best-word`** test cases.

Setup for Scrabble!

In the rest of this problem, you will implement a set of Racket functions to compute the highest-scoring (or "best") Scrabble word, given a particular rack of letters and a list of legal words, **`WL`**.

Here is a function signature to get you started:

```
None
(define (best-word rack WL)
```

`best-word` takes in a string **`rack`** and a list of strings **`WL`**, where each element of **`WL`** is a legal word. It outputs a two-element list in which the first element is the best word and the second element is the score of that best word. If there are ties, **`best-word`** may return any one of the best.

A few examples:

```
None
(check-equal? (best-word "academy" '("ace" "ade" "cad" "cay" "day"))
 '("cay" 8))
(check-equal? (best-word "appler"  '("peal" "peel" "ape" "paper"))
 '("paper" 9))
(check-equal? (best-word "paler"   '("peal" "peel" "ape" "paper"))
 '("peal" 6))
(check-equal? (best-word "kwyjibo" '("ace" "ade" "cad" "cay" "day"))
 '("" 0))
```

Note in the third example, "paper" could not be made because the rack had only a single "p".

In general, there may be more than one best-scoring word. Therefore, we should make sure to run tests similar to the following (that only check the score):

```
None
(check-equal? (second (best-word "bcademy" '("ace" "ade" "cad" "cay" "bay"))) 8)
```

In this case, either "cay" or "bay" would produce that best score. Note that you should not use the procedure **`second`** elsewhere in CS 60 (though you **can** use it anywhere in Scrabble).

You are welcome to design this **`best-word`** function completely on your own. But you should make sure to decompose this computation into a number of small, helper functions. We recommend planning out a strategy for this problem, and **write your test cases** before diving into the coding. Of course, you should write function comments for any helper functions. We also highly recommend that you write test cases for your helper functions since doing so will help you debug your helper functions before you use them. This said, only **`subbag?`** and **`best-word`** need to be "exposed" outside of the definitions file, so tests for the helper functions can go inside the definitions file (meaning the definitions file must require **`rackunit`**).

You are welcome to use any definitions from other parts of this assignment; however, if you choose to do so, *please copy-paste them into* **`scrabble.rkt`**. (For example, in our solution, we used the **`subbag?`** predicate from earlier.) This practice is terrible for software development (what happens if you need to change your definition of a function you copied?), but doing so makes it possible for us to grade each problem separately.

Hints for Scrabble!

Continue reading for suggestions on how to implement **`best-word`**.

We can use an **association-list** (like a Python dictionary) to score information about Scrabble tiles. The following Racket statement defines an association list that provides all letter scores. Copy into your Racket file:

```
None
;; scrabble-tile-bag
;;   letter tile scores and counts from the game of Scrabble
;;   the counts are not needed
;;   obtained from
http://en.wikipedia.org/wiki/Image:Scrabble_tiles_en.jpg
(define scrabble-tile-bag
  '((#\a 1 9) (#\b 3 2) (#\c 3 2) (#\d 2 4) (#\e 1 12)
   (#\f 4 2) (#\g 2 3) (#\h 4 2) (#\i 1 9) (#\j 8 1)
   (#\k 5 1) (#\l 1 4) (#\m 3 2) (#\n 1 6) (#\o 1 8)
   (#\p 3 2) (#\q 10 1)(#\r 1 6) (#\s 1 4) (#\t 1 6)
   (#\u 1 4) (#\v 4 2) (#\w 4 2) (#\x 8 1) (#\y 4 2)
   (#\z 10 1) (#\_ 0 2)) )
;; end define scrabble-tile-bag
;; The underscore is used to represent a blank tile, which is a wild-card
```

Note: You do **not** need to handle the blank tile. We will not test for racks that include blank tiles.

Next, let us introduce the built-in function **`assoc`**. **`(assoc e A)`** takes in an element **`e`** and an association list **`A`** (which contains lists of lists) and returns the element of **`A`** whose own first element is equal to **`e`**. For example:

```
None
(assoc 3 '((0 jan) (1 feb) (2 mar) (3 apr)))   ==>  (3 apr)
(assoc 5 '((1 jan) (2 feb) (3 mar) (4 apr)))   ==>  #f
```

As the second example shows, **`assoc`** returns **`#f`** when **`e`** does not appear as a first element of any of **`A`**'s sublists. You can use **`assoc`** to help look up letters and their scores.

As a possible helper function, construct a function **`score-letter`** that determines the score for its argument, which is a character. For example,

```
None
(check-equal? (score-letter '#\w) 4)
```

Hint: Use **`assoc`**!

As another possible helper function, construct a function **`score-word`** that determines the score for a string of letters. Here are three tests of **`score-word`**:

```
None
(check-equal? (score-word "zzz")  30)
(check-equal? (score-word "fortytwo") 17)
(check-equal? (score-word "twelve")  12)
```

Note that strings in Racket are not the same as symbols, nor are characters the same as strings (as they are in Python). Strings are doubly quoted `"like this"`. The built-in functions **`string->list`** and **`list->string`** convert from strings to lists of characters and vice-versa and may be of use. See them in action:

```
None
(check-equal? (string->list "hi") '(#\h #\i))
(check-equal? (list->string '(#\h #\i)) "hi")
```

Good luck!

Problem 6: Java Arrays

- Learning Goal: Write Java code with arrays.
- Prerequisites: Java conditionals, loops, Arrays; JUnit testing in VSCode
- Starter files: **`JavaArrays.java`** and **`JavaArraysTests.java`** (from JavaBasics_starter**)**
- Submit: Screenshot, and the file **`JavaArrays.java`**

This time, about half of the problems have a "Show Solution", which you are welcome to use. Remember, if you use the "Show Solution" button, please add a comment that describes what insight you gained from looking at the solution. You do not need to add a comment if you use the "Show Hint" button.

The following resources might be helpful:

- Java For and While Loops
  Java Arrays and Loops up to (but not including) "Array Code Patterns"
- The internet (You can Google: your error messages, for explanations, for syntax, etc.)

Write code to pass all the tests

- Complete the methods in **`JavaArrays.java`** to make the tests in **`JavaArraysTests.java`** pass.
  - Your implementation should handle inputs in which arrays are of any size. Note that, in general, you do not have to specially handle different array sizes. If you implement things simply, you should get this functionality for free!
  - To keep things simple this week, there is no need to gracefully handle invalid inputs (i.e. when inputs are of a different size than specified in the doc comment).
- Remember:
  - If you use any of the answers from the CodingBat website, you need to specify in a comment what you learned from looking at the solution code. Remember also that your solutions must have the word `static` in the method signature, but the solutions from CodingBat will not.
  - You are responsible for your code compiling with the provided test cases and for using proper style (except for doc comments) – Go through the checklist before submitting.
- If you get stuck, consult the list of resources (Problem Intro) and debugging suggestions (Part A). If still stuck, consult course staff! We love to help!
- Before you submit, be sure to go over the checklist in How to Submit.

Rubric

#	Name	Functionality	Style	Testing	Total
1	remove-all	1	2	2	5
2	prefix? & sublist?	4	2	4	10
3	enumerate	3	2	2	7
4	superreverse & duperreverse	12	2	2	16
5	subbag? & Scrabble	18	2	2	22
6	Java Arrays	14	6	0	20
		65%	20%	15%	80

Every problem should use recursion and list functions. You may not use higher-order functions, and you may not hard-code outputs tailored to the test cases.

You should aim to pass all test cases for full points.

To receive full points for testing, you must have at least 2 student test cases for each implemented non-helper function. Each of these should test something different about your code (e.g., testing different possible paths through your code).

You should use helper functions and let or let* where appropriate (e.g., in best-word). You will be penalized for all elements of bad style (same as last week) down to a certain minimum. This is a tricky element of CS60 that you may not have experienced before, so we understand that it can be difficult. It is a good idea to do a detail oriented review of your code for style before turning it in!