Quiz 1: Practice Problems

CS 60 • Fall 2025

Note: To mimic a quiz environment, we recommend that you work on the following practice problems on pencil and paper **without opening DrRacket or VS Code**.

Basics and Recursion

1. Evaluate the following Racket expression:

```
(let ([x 5])
    (let ([x 2]
          [y x])
      (list y x)))
```

   Source: https://docs.racket-lang.org/reference/let.html

Output:

   Consider changing the second `let` to `let*`. Does the output change, and if so, what is the new output?

Output:

2. If we wanted to reverse a Racket list, we could do so using the (correct!) method **`reverse`** shown at left below. Someone wrote an incorrect version of reverse (below, right) which we are calling badReverse. In particular, note that the last line of reverse has been changed:

```
(define (reverse L)
  (if (empty? L)
      L
      (append (reverse (rest L))
              (list (first L)))))
```

```
(define (badReverse L)
  (if (empty? L)
      L
      (append (badReverse (rest L))
              (list L))))
```

Recall that, especially when writing recursive code, it can be helpful to examine procedure calls by using trace. For example, let's say we trace reverse on the expression

```
> (reverse '())
```

Then, Racket would print out:

```
> (reverse '())
<'()
'()
```

The above output is Racket's way of displaying function calls (denoted by >) and outputs (denoted by <). Here, calling `(reverse '())` returns output `'()` and we can see that there were no recursive calls. The last line tells us the output of the expression (and is not part of the trace).

Here's a slightly more complicated example. Let's say we trace the expression

```
> (reverse '(2 3))
```

Then, Racket would print out:

```
>(reverse '(2 3)) ; the expression we're tracing
> (reverse '(3))  ; successive (recursive) call to reverse
> >(reverse '())  ; successive (recursive) call to reverse
< <'()            ; evaluated base case
< '(3)            ; apply combiner to result of recursive call
<'(3 2)           ; apply combiner to result of recursive call
'(3 2)            ; final output
```

Conduct a similar trace on this call to `badReverse`. You do not need to include comments.

```
> (badReverse '(2 3))
```

Source: CS 60 Spring 2019, Final

```
> (badReverse '(2 3))
```

3. We have a triangle made of blocks. The topmost row has 1 block, the next row down has 2 blocks, the next row has 3 blocks, and so on. Compute recursively (no loops or multiplication) the total number of blocks in such a triangle with the given number of rows.

```
(triangle 0) → 0
(triangle 1) → 1
(triangle 2) → 3
```

Source: https://codingbat.com/prob/p194781

```
(define (triangle n)




)
```

4. Given a non-negative int n, return the sum of its digits recursively (no loops). Note that modulo by 10 yields the rightmost digit, e.g. `(modulo 126 10)` → `6`, while dividing by 10 removes the rightmost digit `(quotient 126 10)` → `12`.

```
(sum-digits 126) → 9
(sum-digits 49) → 13
(sum-digits 12) → 3
```

Source: https://codingbat.com/prob/p163932

```
(define (sum-digits n)









)
```

Lists

5. Examine this (partial) code and its output:

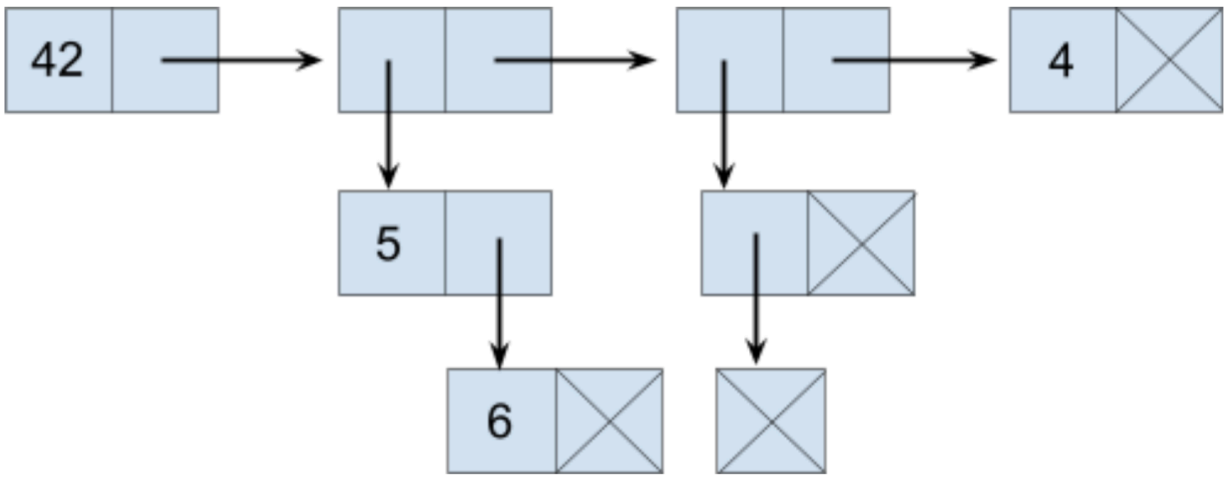
```
> (__A__ '(1 2) '(3))
'((1 2) 3)

> (__B__ (__C__ 1 2) '(3)))
'(1 2 3)
```

   Fill in the blank for A, B, and C.

   Source: New problem, inspired by CS 60 Spring 2020, CC 3

A	
B	
C	

6. Suppose the following linked list was assigned to a variable **L**. How would Racket output **L**? That is, what would result from typing the following into the Racket interpreter: **> L**



   Source: CS 60 Fall 2021, Optional CC

Output:

7. Given an list of ints, is it possible to choose a group of some of the ints, such that the group sums to the given target? This is a classic backtracking (use-it-or-lose-it) recursion problem. Once you understand the recursive backtracking strategy in this problem, you can use the same pattern for many problems to search a space of choices. No loops are needed -- the recursive calls progress down the list.

```
(group-sum '(2 4 8) 10) → true
(group-sum '(2 4 8) 14) → true
(group-sum '(2 4 8) 9) → false
```

Source: https://codingbat.com/prob/p145416

```
(define (group-sum L target)









)
```

HOFs

8. Given a list of non-negative integers, return an integer list of the rightmost digits. (Note: Use modulo.)

```
(right-digits '(1 22 93)) → '(1 2 3)
(right-digits '(16 8 886 8 1)) → '(6 8 6 8 1)
(right-digits '(10 0)) → '(0 0)
```

Source: https://codingbat.com/prob/p145416

```
(define (right-digits L)




)
```

9. Given a list of non-negative integers, return a list of those numbers multiplied by 2, omitting any of the resulting numbers that end in 2.

```
(two2 '(1 2 3)) → '(4 6)
(two2 '(2 6 11)) → '(4)
(two2 '(0)) → '(0)
```

Source: https://codingbat.com/prob/p148198

```
(define (two2 L)




)
```

10. Let's say you have a budget of $30 for small purchases ($5 or less). Given a list of purchases, you want to know how much of your small-purchases budget you have remaining.

```
(define budget 30)
(define purchases '(2 5 47 36 8 11 4 17 3))

(foldr (lambda ...)
       budget
       purchases)

;; the output of the above call is:
> 16
```

From the options below, select the **`lambda`** expression(s) that, if substituted into the above call to **`foldr`**, would successfully compute the funds remaining in your $30 budget:

☐
```
;; A
(lambda (x y)
   (if (<= x 5) (- y x)
                y))
```

☐
```
;; B
(lambda (a b)
   (if (> a 5) b
               (+ a b)))
```

☐
```
;; C
(lambda (price fundsRemaining)
   (if (> price 5) fundsRemaining
                   (- fundsRemaining price)))
```

Source: CS 60 Fall 2021, Optional CC

Additional practice: Add function-level comments and additional test cases to the above problems.

- A function-level comment should include a description of the function and its inputs and outputs.
- Test cases should have the correct input-output pair. You should test simpler cases that help debugging, tricky cases, and all possible cases through the code (e.g. all branches for if/cond, all base cases for recursion).

Test cases / unit tests and Big O

11. Consider the following code:

```
public class Iterative {
  public static boolean f(int[] values) {
      for (int i = 0; i < values.length; i++) {
        if (values[i] % 2 == 0) {
          return true;
        }
      }
      return false;
  }
}
```

A. Write tests for this code, by filling in the blanks below.

```
/**
  * Test base case, when the array of values is 0
  */
 @Test
 public void test1() {
     int[] testValues =          ;
     assertFalse(             );
 }

 /**
  * Test base case, when the first element of the array is even
  */
 @Test
 public void test2() {
     int[] testValues =          ;
     assertTrue(             );
 }

 /**
  * Test case with value that eventually returns true
  */
 @Test
 public void test3() {
     int[] testValues =          ;
     assertTrue(             );
 }

 /**
  * Test case with a value that eventually returns false
  */
 @Test
 public void test4() {
     int[] testValues =          ;
     assertFalse(             );
 }
```

The code is repeated for reference here:

```
public class Iterative {
  public static boolean f(int[] values) {
      for (int i = 0; i < values.length; i++) {
        if (values[i] % 2 == 0) {
          return true;
        }
      }
      return false;
  }
}
```

B. Use loop counting to analyze how many times the code performs the modulo operation (%), in the worst case. Show your work for arriving at the summation (i.e., the sequence of numbers that would be added together), the total for the summation, and the Big O runtime. Be sure to define any variables you use to describe the performance!

Note: HW 2 is a great resource for more loop-counting problems. It would be good to review it!

The code is repeated for reference here:

```
public class Iterative {
  public static boolean f(int[] values) {
      for (int i = 0; i < values.length; i++) {
        if (values[i] % 2 == 0) {
          return true;
        }
      }
      return false;
  }
}
```

C. Consider the recursive implementation of the same implementation, below. Then, write a recurrence relation that describes the number of times the code performs the modulo operation (%). Unroll the recurrence to get a summation of values, a total, and a Big O description. Be sure to define any variables you use to describe the performance!

```
public static boolean f(List<Integer> values) {
      if (values.isEmpty()) {
        return false;
      }

      if (values.get(0) % 2 == 0) {
        return true;
      }

      return Recursive.f(values.subList(1, values.size()));
}
```

Solutions

Note that there may be multiple ways of implementing the same functionality.

1. `'(5 2)`
   If we change to let*, then the result is `'(2 2)`

2.

```
> (badReverse '(3))
> >(badReverse '())
< <'()
< '((3))
<'((3) (2 3))
```

3.

```
(define (sum-digits n)
  (if (< n 10)
      n
      (+ (modulo n 10)
         (sum-digits (quotient n 10))))
  )
```

4.

```
(define (triangle n)
  (if (= n 0)
      0
      (+ n
         (triangle (- n 1))))
  )
```

5. `cons`, `append`, `list`

6. `'(42 (5 6) (()) 4)`

7.

```
(define (group-sum L target)
  (if (empty? L)
      ; base case: if there are no numbers left,
      ; then there is a solution only if target is 0
      (= target 0)

      ; key idea: first element is chosen or it is not
      ; deal with first element, let recursion deal with rest of list
      ; use-it: use element, subtract from target in recursive call
      ; lose-it: lose element, target stays same
      (or (group-sum (rest L) (- target (first L)))
          (group-sum (rest L) target))
  ))
```

Alternative:

```
(define (group-sum L target)
  (cond
    ; base case: if there are no numbers left,
    ; then there is a solution only if target is 0
    [(empty? L)
     (= target 0)]

    ; key idea: first element is chosen or it is not
    ; deal with first element, let recursion deal with rest of list

    ; use-it: use element, subtract from target in recursive call
    [(group-sum (rest L) (- target (first L)))
     #t]

    ; lose-it: lose element, target stays same
    [(group-sum (rest L) target)
     #t]

    ; if neither of above worked, it is not possible
    [else #f]
  ))
```

8.

```
(define (right-digits L)
  (map (lambda (x) (modulo x 10))
       L))
```

9.

```
(define (two2 L)
  (filter
   (lambda (x) (not (= (modulo x 10) 2)))
   (map
    (lambda (x) (* x 2))
    L)))
```

10. A, C

11.

a.

```
/**
  * Test base case, when the array of values is 0
  */
 @Test
 public void test1() {
     int[] testValues = {};
     assertFalse(Iterative.f(testValues));
 }

 /**
  * Test base case, when the first element of the array is even
  */
 @Test
 public void test2() {
     int[] testValues = {2};
     assertTrue(Iterative.f(testValues));
 }

 /**
  * Test case with a value that eventually returns true
  */
 @Test
 public void test3() {
     int[] testValues = {1, 3, 4};
     assertTrue(Iterative.f(testValues));
 }

 /**
  * Test case with a value that eventually returns false
  */
 @Test
 public void test4() {
     int[] testValues = {1, 3, 5};
     assertFalse(Iterative.f(testValues));
 }
```

b.

Let N be the length of the input array.

i	Number of % for this value of i
0	1
1	1
…	…
N-1	1

Total = 1 + 1 + … + 1 = N, which is O(N)

c.

Let N be the length of the input list.

Recurrence relation:
T(N) = 0 for all N <= 0
T(N) = 1 + T(N-1) for all N > 0

Unrolling:
T(N) = 1 + T(N-1)
T(N) = 1 + 1 + T(N - 2)
T(N) = 1 + 1 + … + 1 + T(N - N)

Finding a pattern, and a total sum:
T(N) = N + T(N-N)
= N + 0
= N

Big O:
O(N)