Name: ________________________________

CS4800: Algorithms and Data

Exam 3: Practice

Exam Policies. Your work on this exam must be your own. You *may not* copy solutions from other students. You *may not* communicate with other students during the exam. You *may not* communicate with anyone outside the class during the exam.

This exam is open book, open notes. You *may* refer to the text, prior homeworks, past exams, practice exams, your own notes, and anything else you brought with you on paper. You *may not* share notes or any other written materials with other students during the exam. You *may not* use electronic devices during the exam.

The contents of the exam are confidential. You *may not* discuss the contents of the exam with any student who has not yet taken the exam. Due to illness, travel, or other special circumstances, some students may take exams at different times than others.

You *must* complete the exam in the time provided. Your exam *will not* be accepted if you work past the deadline.

If you violate any of these rules, you will receive a zero as your grade for this exam and you will be reported to the Office of Student Conduct and Conflict Resolution (OSCCR).

Problem A. Consider the *Tower of Hanoi* puzzle:

> There are three pegs and n disks. The disks each have a different radius, and a hole through their center so they can be placed over the pegs. To begin with, all n disks are stacked on the leftmost peg, ordered by width so that the widest is at the bottom and the narrowest is at the top.
>
> The goal of the puzzle is to move all n disks to the rightmost peg. You can only move one peg at a time, from one peg to any other. You can only place disks on top of larger disks; you can never have a large disk on top of a smaller one.
>
> There is a recursive solution to the puzzle, based on the fact that for any $m \le n$ you can always move the smallest m disks around without regard for the larger ones, because the smaller ones can be freely placed on top of the larger ones.
>
> To solve the puzzle for n disks, first move $n-1$ disks to the middle peg. This gets them out of the way so you can move the last disk. Move the last disk to the rightmost peg. Finally, move the $n-1$ disks from the middle to the rightmost peg.
>
> At zero pegs, of course, you don't have to do anything. And note that for the recursive steps, the goal changes. At n disks, for instance, you needed to move from the left to the right, using the middle peg as a placeholder. The first time you move $n-1$ disks, they move from the left to the middle, and the rightmost peg is free. The second time, they move from the middle to the right, and the left is free.

The function function `tower : Nat -> Moves` that produces a sequence of moves to solve the Tower of Hanoi puzzle. Here is the data definition for `Moves`:

```
;; A Moves is either a Move or a (Listof Moves),
;; where the Moves in a list are ordered left to right.

;; A Move is (move Peg Peg),
;; meaning you take the topmost disk of the from peg
;; and place it on top of the to peg.
(struct move [from to])

;; A Peg is 'left, 'right, or 'middle.
```

The `Moves` data structure is ordered from left to right; nested lists make it easier to combine `Moves` values without using `append`, but otherwise don't matter. For example, `(tower 2)` might produce any of the following results, among many other possibilities:

```
(list (move 'left 'middle) (move 'left 'right) (move 'middle 'right))
(list (list (move 'left 'middle) (move 'left 'right)) (move 'middle 'right))
(list (move 'left 'middle) (list (move 'left 'right) (move 'middle 'right)))
```

A.1 Write the function `tower` using the recursive strategy described above. The main recursion will need to be done in a helper function.

Solution:

```
(define (tower n)
  (tower/pegs n 'left 'middle 'right))

(define (tower/pegs n from spare to)
  (cond
    [(= n 0) empty]
    [(= n 1) (move from to)]
    [else
     (list
       (tower/pegs (sub1 n) from to spare)
       (move from to)
       (tower/pegs (sub1 n) spare from to))]))
```

A.2 State the running time of `tower` as a recurrence.

Solution: $T(n) = 2T(n-1) + \Theta(1)$

A.3 Solve the recurrence. Use the master method if possible; otherwise, state why the master method does not apply. You must find a simplified Θ-form solution either way.

Solution: The master method does not apply, as the recurrence is not in the form $T(n) = aT(\frac{n}{b}) + f(n)$. Nevertheless, each step is essentially double the last, so the solution is $T(n) = \Theta(2^n)$.

A.4 How many *different* recursive subproblems does `(tower n)` need to solve, at most?

Solution: There argument n can take on $n+1$ different values: any integer from 0 to n. The arguments $from$, to, and $spare$ can take on six different sets of values, depending on the current source and destination:

1. `'left`, `'middle`, `'right`
2. `'left`, `'right`, `'middle`
3. `'middle`, `'left`, `'right`
4. `'middle`, `'right`, `'left`
5. `'right`, `'left`, `'middle`
6. `'right`, `'middle`, `'left`

Therefore there are at most $6n+6$ distinct subproblems that `(tower n)` might need to solve.

A.5 Write a *memoized* version of tower that records the solution to each subproblem.

Solution:

```
(define (tower n)
  (define table (make-vector (+ (* 6 n) 6) #false))
  (tower/pegs table n 'left 'middle 'right))

(define (tower/pegs table n from spare to)
  (define index (table-index n from spare to))
  (define entry (vector-ref table index))
  (cond
    [entry entry]
    [else
     (define result
       (cond
         [(= n 0) empty]
         [(= n 1) (move from to)]
         [else
          (list
            (tower/pegs table (sub1 n) from to spare)
            (move from to)
            (tower/pegs table (sub1 n) spare from to))]))
     (vector-set! table index result)
     result]))

(define (table-index n from spare to)
  (+ (* n 6) (table-offset from spare to)))

(define (table-offset from spare to)
  (cond
    [(equal? from 'left)
     (cond
       [(equal? to 'middle) 0]
       [(equal? to 'right) 1])]
    [(equal? from 'middle)
     (cond
       [(equal? to 'left) 2]
       [(equal? to 'right) 3])]
    [(equal? from 'right)
     (cond
       [(equal? to 'left) 4]
       [(equal? to 'middle) 5])]))
```

A.6 State the running time of the memoized version of `tower`.

Solution: Since there are $6n+6$ subproblems that take $\Theta(1)$ time each, the memoized version runs in $O(n)$ time.

Problem B. Consider the adjacency-list representation for a graph:

```
;; A Graph is a (Vectorof (Listof Vertex)),
;; where each vertex is an index into the graph.

;; For instance, consider the graph:
;;
;; 0 -> 1
;; |    ^
;; v    |
;; 3 -> 2
;;
;; The graph's representation would be:
(vector
  (list 1 3)
  (list)
  (list 1)
  (list 2))
```

B.1 Write a program `cycle? : Graph -> Boolean` that produces `#true` if the graph contains a cycle and `#false` if it does not. (*Note:* you have space to continue on the next page if you need to.)

Solution:

```
;; This solution is an adaptation of Depth-First Search that tracks the state
;; of each vertex as it goes (unseen, visiting, or seen), and detects a cycle
;; whenever it rediscovers a vertex that it is currently visiting.

;; A State is a (Vectorof Status).
;; A Status is 'unseen, 'visiting, or 'seen.

;; cycle? : Graph -> Boolean
(define (cycle? G)
  (cycle-at-or-after? G 0 (make-vector (vector-length G) 'unseen)))

;; cycle-at-or-after? : Graph Nat State -> Boolean
(define (cycle-at-or-after? G i state)
  (cond
    [(>= i (vector-length G)) #false]
    [else (or (cycle-at? G i state)
              (cycle-at-or-after? G (add1 i) state))]))

;; Continued on next page ...
```

Solution:

```
;; ... and resumed from previous page.

;; cycle-at? : Graph Nat State -> Boolean
(define (cycle-at? G i state)
  (define status (vector-ref state i))
  (cond
    [(equal? status 'visiting) #true]
    [(equal? status 'seen) #false]
    [(equal? status 'unseen)
     (vector-set! state i 'visiting)
     (define result-for-neighbors
       (cycle-at-any? G (vector-ref G i) state))
     (vector-set! state i 'seen)
     result-for-neighbors]))

;; cycle-at-any? : Graph (Listof Nat) State -> Boolean
(define (cycle-at-any? G is state)
  (cond
    [(empty? is) #false]
    [else (or (cycle-at? G (first is) state)
              (cycle-at-any? G (rest is) state))]))
```

B.2 State and justify the running time of `cycle?`.

Solution: The function `cycle-at?` is called once for each vertex, by `cycle-at-or-after?`, plus once for each edge, by `cycle-at-any?`. Other than iterating through vectors and lists to make each of these calls, each function does $O(1)$ work. Therefore the function runs in $O(V+E)$ time, just like the Depth-First Search from which it is adapted.