CMSC 15100 Introduction to Computer Science I

The University of Chicago, Summer 2020

Final Examination

Please write your name here:

Yifei Li

Instructions:

- Due date is July 26 11:59 pm.
- The exam is open-book and open-internet. The rules are same as that of the midterm.
- For the problems that ask you to write Racket code, it's more important that your code has the right ideas than that it compiles. Try and use good coding style if possible (indentation and new lines, correct parens, meaningful variable names).
- You may freely use any functions seen in class. Write helper functions as you see fit.
- There are total 13 questions. Questions 9, 10, 11, 12 and 13 are extra-credit.
- Do the **easy / familiar questions** before attempting the difficult ones.
- You don't have to write any tests (`check-expect`, etc.) anywhere on the exam, nor do you have to `require` anything. You don't have to write any comments either, but feel free to add comments if you think your code would be easier to understand.

This box is for grading. Please continue to the next page.

Problem	Points	Score
1	12	
2	8	
3	7	
4	10	
5	15	
6	9	
7	11	
8	10	
9(EC)	9	
10(EC)	8	
11 (EC)	8	
12 (EC)	7	
13 (EC)	10	
Total	82	

1. (2 points each)

(a) What is a higher-order function?

A higher order function is a function that takes functions as inputs and outputs.

(b) What is a "polymorphic function"?

A polymorphic function is a function that is able to take in data with multiple types as input, process these data in different ways according to their types, and give outputs in different types.

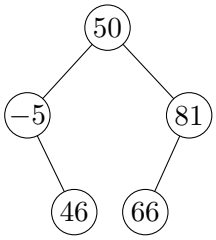
(c) What is "mutual recursion"?

Mutual recursion is a recursion relationship that two recursion types/functions that rely on each other. For example, the return value of func1 calls func2, and vice versa.

(d) What does the term degree of a vertex in a graph mean?

The degree of vertex in graph is the number of edges that connected to that vertex.

(e) Define a variable named `tree` for this `(BST Integer)`.




```
(define-struct (BST A)
  ([lte? : A A -> Boolean]
   [bst : (U 'none (Node A))]))

(define-struct (Node A)
  ([value : A]
   [left-child : (U 'none (Node A))]
   [right-child : (U 'none (Node A))]))
```

```
(: tree : (BST Integer))
(define tree
  (BST
   integer-lte?
   (Node 50
             (Node -5 'none (Node 46 'none 'none))
             (Node 81 (Node 66 'none 'none) 'none))))
```

(f) What is the "BST invariant"?

For every node of the BST tree, all nodes in the left subtree of the node have value smaller than the value of that node, all nodes in the right subtree of the node have value greater than the value of that node.

2.

(a) **(3 points)** The *third moment* of a list of numbers $x_1, x_2, \ldots, x_n$ is

$$\frac{x_1^3 + x_2^3 + x_3^3 + \cdots + x_n^3}{n}$$

Write a function `third-moment` which computes the third moment of a list of numbers. Your code can behave arbitrarily if the input list is empty.

```
(: third-moment : (Listof Real) -> Real)
```

```
(: third-moment : (Listof Real) -> Real)
(define (third-moment lst)
  (local
    {
     (: element : Real → Real)
     (define (element r)
       (/ (expt r 3) (length lst)))}
    (foldr + 0 (map element lst))))
```

(b) **(5 points)** The power set of a set is defined as the set of all subsets of that set. For example the power set of the set $\{1, 2\}$ is the set $\{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$ here $\emptyset$ represents the empty set. For this problem we will represent sets using lists. You can assume your list will have no duplicate elements. Write a racket function which takes a set of integers as input (represented in the form of a list of integers) and output its power set (represented in the form of list of list of integers). Calling

```
(power-set (list 1 2 3))
```

should give the following output

```
'((1 2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())
```

The order of numbers or elements in the list is not important. The empty list () represents the empty-set. (Hint: The easiest way to do is to use recursion and map. Shouldn't take more than 5-6 lines of code. You don't need to worry about the case where input is the empty set)

```
(: power-set : (Listof Integer) -> (Listof (Listof Integer)))
```

```
(: power-set : (Listof Integer) → (Listof (Listof Integer)))
(define (power-set lst)
  (local
    {
     (: append-first : (Listof Integer) → (Listof Integer))
     (define (append-first my-lst)
       (append (list (first lst)) my-lst))}
    (cond
      [(= (length lst) 1) (list (list) lst)]
      [else (append (map append-first (power-set (rest lst)))
                    (power-set (rest lst)))])))
```

3. The Earth has four hemispheres, and any location on Earth lies in exactly two of them,

```
(define-struct Hemisphere
  ([longitude : (U 'east 'west)]
   [latitude : (U 'south 'north)]))
```

(a) **(3 points)** Make a function `country->hemisphere` which takes in a country name as a `String` and outputs a `Hemisphere` for that country's location. Use `match`. Your function doesn't have to work for every country, just make it work for at least 3 different countries.

```
(: country->hemisphere : String -> Hemisphere)
```

```
(: country→hemisphere : String -> Hemisphere)
(define (country→hemisphere str)
  (match str
    ["United States" (Hemisphere 'west 'north)]
    ["China" (Hemisphere 'east 'north)]
    ["Japan" (Hemisphere 'east 'north)]
    ["Australia" (Hemisphere 'east 'south)]))
```

(b) **(4 points)** Write the function `same-hemisphere?` which takes two countries and checks if they're in the same `Hemisphere` (both latitude and longitude). Use `match`, and don't use any selector functions.

```
(: same-hemisphere? : String String -> Boolean)
```

```
(: same-hemisphere? : String String -> Boolean)
(define (same-hemisphere? str1 str2)
  (match* ((country→hemisphere str1) (country→hemisphere str2))
    [(n n) #t]
    [(a b) #f]))
```

4. Phil the Phoenix is the mascot of UChicago, and like any phoenix, no matter how many times he dies, he always rises from the ashes and comes back to life. In Racket we represent Phil by a recursive type,

```
(define-struct Phil
  ([days-alive : Integer]
   [past-life : (U 'none Phil)]))
```

(a) **(2 points)** What functions are automatically defined after using the above `define-struct`?

```
(Phil Integer (U 'none Phil) -> Phil)
(Phil-days-alive Phil -> Integer)
(Phil-past-life Phil -> (U 'none Phil))
(Phil? Any -> Boolean)
```

(b) **(3 points)** Write a function `rebirth` which creates a new Phil who has been alive zero days. This Phil should have the input Phil as its past life.

```
(: rebirth : Phil -> Phil)
```

```
(: rebirth : Phil -> Phil)
(define (rebirth phil)
  (Phil 0 phil))
```

(c) **(5 points)** Write a function `total-days-alive` that computes how many total days Phil has been alive, across all his past lives.

```
(: total-days-alive : Phil -> Integer)
```

```
(: total-days-alive : Phil -> Integer)
(define (total-days-alive phil)
  (match (Phil-past-life phil)
     ['none (Phil-days-alive phil)]
     [(Phil d p) (+ (total-days-alive (Phil d p))
                    (Phil-days-alive phil))]))
```

5. DNA sequences consist of long lines of nitrogenous bases, each of which are either A, C, T, or G. For the purposes of this problem, we define the following types for DNA,

```
(define-type Base (U 'A 'C 'T 'G))
(define-type DNA (Vectorof Base))
```

Two DNA sequeces can intertwine if they are *complementary*: everywhere the first sequence has A, the second sequence has T and vice versa, and everywhere the first sequence has C, the second sequence has G and vice versa. These are two complementary sequences,

```
A T G T G C C A C A G G A T A A G T T A A
T A C A C G G T G T C C T A T T C A A T T
```

(a) **(3 points)** Make a function `complementary-bases?` which takes in two `Base`s, and outputs whether or not the two bases are complementary.

```
(: complementary-bases? : Base Base -> Boolean)
```

```
(: complementary-bases? : Base Base -> Boolean)
(define (complementary-bases? b1 b2)
  (match* (b1 b2)
    [('A 'T) #t]
    [('T 'A) #t]
    [('C 'G) #t]
    [('G 'C) #t]
    [(_ _) #f]))
```

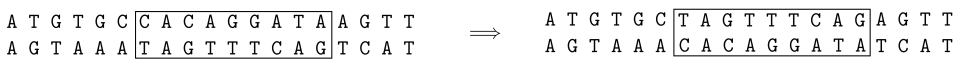
(b) **(6 points)** Make a function that checks if two DNA sequences are complementary. You can assume the two sequences have the same length.

```
(: complementary? : DNA DNA -> Boolean)
```

Hint: you will probably need a local helper function

```
(: complementary? : DNA DNA -> Boolean)
(define (complementary? d1 d2)
  (local
    {
     (: get-complementary : Base -> Base)
     (define (get-complementary b)
       (match b
         ['A 'T]
         ['T 'A]
         ['C 'G]
         ['G 'C]))
     (: dna=? : DNA DNA Integer -> Boolean)
     (define (dna=? d1 d2 i)
       (cond
         [(= i (vector-length d1))
             (if (symbol=? (vector-ref d1 (- i 1))
                           (vector-ref d2 (- i 1))) #t #f)]
         [else (dna=? d1 d2 (+ 1 i))]))
     }
    (dna=? d2 (vector-map get-complementary d1) 0)))
```

(c) **(6 points)** Crossing over is a phenomenon in genetics where two DNA sequences (not necessarily complementary) swap inside a certain region (in this example, the boxed region),



Make a function `cross-over!` which takes in four inputs: the two DNA sequences, and the start and end index of the region, and swaps the contents of the DNA sequences between start and end (both inclusive). This function should be imperative and have return type `Void`.

```
(: cross-over! : DNA DNA Integer Integer -> Void)
```

```
(: cross-over! : DNA DNA Integer Integer -> Void)
(define (cross-over! d1 d2 start end)
  (local
    {
     (: swap! : DNA DNA Integer -> Void)
     (define (swap! d1 d2 i)
       (local
         {
          (: temp : Base)
          (define temp (vector-ref d1 i))}
         (begin
           (vector-set! d1 i (vector-ref d2 i))
           (vector-set! d2 i temp)
           (cond
             [(= i end) (void)]
             [else (swap! d1 d2 (+ i 1))])))))}
     (swap! d1 d2 start)))
```

6. (4+5 points) Fill in the blanks in these implementations of the functions `map` and `andmap`,

```
(: map : ___(All (A) (A -> A) (Listof A) -> (Listof A))____)

(define (map f my-list)

  (cond

    [(empty? my-list) _____'()_________]

    [else (cons   _(f (first my-list))_     (map ___f____(rest my-list)))]))
```

```
(: andmap : (All (A) (A -> Boolean) (Listof A) -> Boolean)_)

(define (andmap f? my-list)

  (cond

    [(empty? my-list) _#t_____________]

    [else (_ and_____    (_____f?_____        __(first my-list)__)

                         (andmap   ____f?____     __(rest my-list)___))]))
```

7. This problem will work with the types we developed for the checkers game,

```
(define-type Color (U 'red 'black))

(define-struct Loc
  ([row : Integer]
   [col : Integer]))

(define-struct Piece
  ([color : Color]
   [loc : Loc]))

(define-struct Checkers
  ([pieces : (Listof Piece)]
   [turn : Color]
   [clicked-piece : (U 'none Piece)]))
```

(a) **(4 points)** Make a function `change-turn` that changes the turn of the game from `'red` to `'black` or from `'black` to `'red`.

```
(: change-turn : Checkers -> Checkers)
```

```
(: change-turn : Checkers → Checkers)
(define (change-turn game)
  (match game
    [(Checkers lst 'red cli) (Checkers lst 'black cli)]
    [(Checkers lst 'black cli) (Checkers lst 'red cli)]))
```

(b) **(7 points)** Make a function which takes in a `Checkers` game and a `Loc`, and returns true if the `Loc` is two squares diagonally away from the current `clicked-piece` (in any of the four diagonal directions).

It doesn't matter what your code does when `clicked-piece` is `'none`; however, your code should not cause the common type error that occurs when you assume `clicked-piece` is a `Piece`.

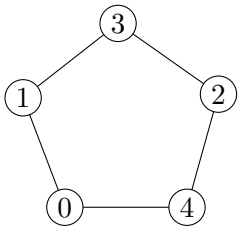
```
(: two-squares-away? : Checkers Loc -> Boolean)
```

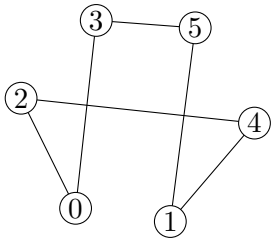
```
(: two-squares-away? : Checkers Loc → Boolean)
(define (two-squares-away? game loc)
  (match (Checkers-clicked-piece game)
    [(Piece _ piece-loc)
     (and
      (or (= 2 (- (Loc-row piece-loc) (Loc-row loc)))
          (= -2 (- (Loc-row piece-loc) (Loc-row loc))))
      (or (= 2 (- (Loc-col piece-loc) (Loc-col loc)))
          (= -2 (- (Loc-col piece-loc) (Loc-col loc)))))]))
```

8. Recall the types for graphs,

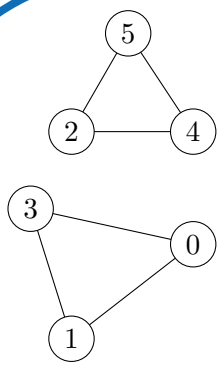
```
(define-type Vertex Integer)
(define-struct Graph
  ([n : Integer]
   [adj : (Vectorof (Listof Vertex))]))
```

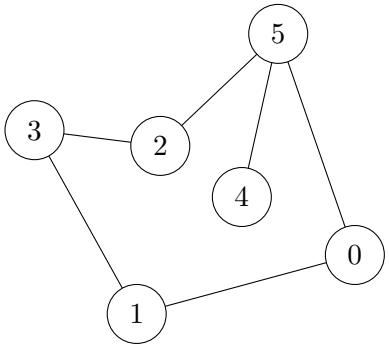
A graph is a *cycle* if it is connected and every vertex has degree 2. For example, these graphs are cycles,





These graphs are not cycles,





You may find some of the following functions from class useful,

```
(: connected? : Graph -> Boolean)
;; checks if a graph is connected
;; runtime is O(m + n)

(: degree : Graph Vertex -> Integer)
;; computes the degree of the given vertex
;; runtime is O(degree of the vertex)

(: all-vertices : Graph -> (Listof Vertex))
;; create a list of all vertices of the graph
;; runtime is O(n)
```

(a) **(2 points)** Define a variable for one of the graphs on the previous page. Circle the graph you chose.

```
(: graph2 : Graph)
(define graph2 (Graph 6 (vector '(1 3) '(0 3) '(4 5) '(0 1) '(2 5) '(2 4))))
```

(b) **(5 points)** Write a function `cycle?` which checks if the input graph is a cycle.

```
(: cycle? : Graph -> Boolean)
```

```
(: cycle? : Graph → Boolean)
(define (cycle? g)
  (local
    {
     (: two-deg? : Vertex → Boolean)
     (define (two-deg? v)
       (match (degree g v)
         [2 #t]
         [_ #f]))}
    (and
     (connected? g)
     (andmap two-deg? (all-vertices g)))))
```

(c) **(3 points)** Analyze the runtime of your `cycle?` function in terms of the number of vertices n and the number of edges m of the input graph.

9.(Extra Credit) The function `unique?`, which checks if a list of integers has no duplicates, is implemented recursively below using a helper function `in?`,

```
(: in? : (Listof Integer) Integer -> Boolean)
;; checks if target is in my-list
(define (in? my-list target)
  (cond
    [(empty? my-list) #f]
    [else (or (= (first my-list) target)
              (in? (rest my-list) target))]))

(: unique? : (Listof Integer) -> Boolean)
;; checks if a list has no duplicates
(define (unique? my-list)
  (cond
    [(empty? my-list) #t]
    [else (and (not (in? (rest my-list) (first my-list)))
               (unique? (rest my-list)))]))
```

(a) **(2 points)** Write a recurrence for $T_{in}(n)$, the number of operations that occur when `in?` is run on a list of length n. Explain the different terms of the recurrence.

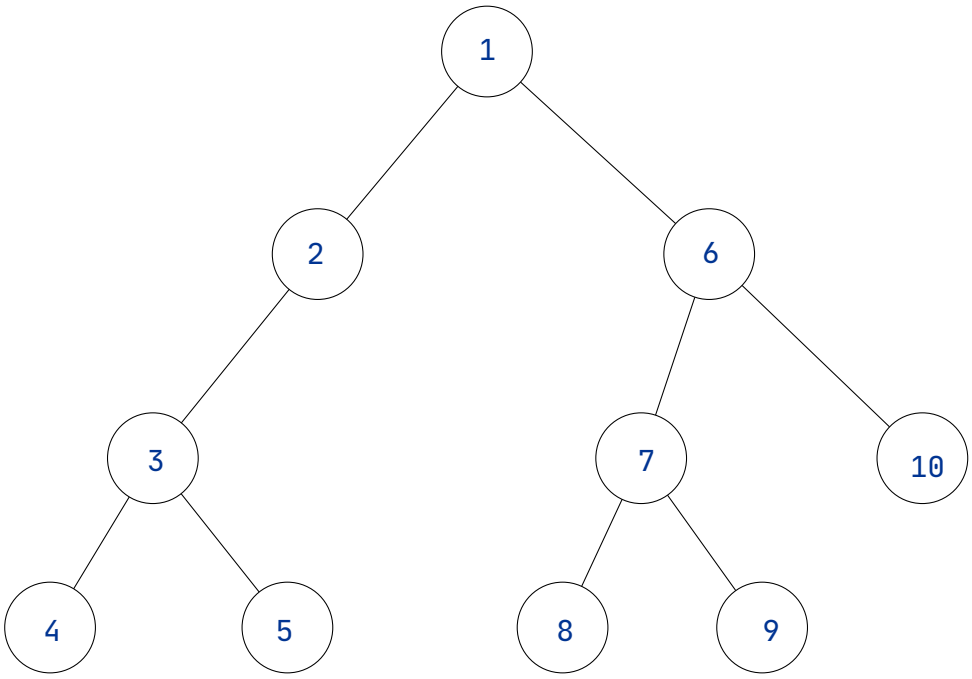
(b) **(2 points)** Solve your recurrence from part (a) using the tree method.

(c) **(3 points)** Write a recurrence for $T(n)$, the number of operations for `unique?` on an input of length n. Explain the different terms of the recurrence.

(d) **(2 points)** Solve your recurrence from part (c) using the tree method.

10.(Extra Credit) You may have noticed that our pictures for binary trees and for graphs look very similar (circles with lines between them). We can convert a tree to a graph by taking a vertex for each node in the tree, and an edge between any parent and child node.
Once we do this, let's say we run depth-first search from the root node. What order will the vertices/nodes of the tree be explored? This order depends on the order of the edges in each vertex's adjacency list – assume that the edges are always ordered left child, right child, parent (not all of these will exist for every vertex). Here we are treating your trees as **undirected** graphs. That's why there is an edge to parent as well.

(a) **(3 points)** For the binary tree pictured below, what order will the vertices be explored if we run DFS from the root? Number the vertices from 1 to 10.



The next parts use the type `(Tree A)`,

```
(define-struct (Tree A)
  ([value : A]
   [left-child : (U 'none (Tree A))]
   [right-child : (U 'none (Tree A))]))
```

(b) **(5 points)** Without using `dfs!`, write a function `dfs-order` that takes a `(Tree A)` or `'none` and outputs a `(Listof A)` of the values in the tree, ordered by the DFS-from-root exploration order. Your function should use recursion. (Hint: This function was already covered in class while discusing tree traversal algorithms. It is equivalent to one of these algorithms pre-order, in-order , post-order. Which one is it? Your solution to part a) should help you answer it)

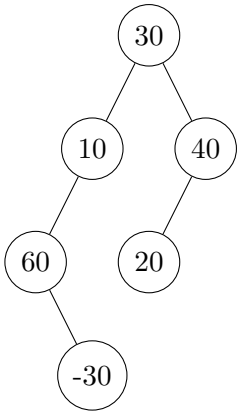
```
(: dfs-order : (All (A) (U 'none (Tree A)) → (Listof A)))
(define (dfs-order tree)
  (match tree
    ['none (list)]
    [(Tree v l r) (append (list v) (dfs-order l) (dfs-order r))]))
```

11. (Extra Credit) (8 points) Write a racket function takes a binary tree as input outputs the the sum of all nodes on the longest path from root to leaf node. If two or more paths compete for the longest path, then the path having maximum sum of nodes is being considered. Feel free to use height function discussed in the class to compute heights of binary trees.

You can use the type `Tree`,

```
(define-struct Tree
  ([value : Integer]
   [left-child : (U 'none Tree)]
   [right-child : (U 'none Tree )]))
```

For example if the following tree is given as a input to your program your output should be 70



12. (Extra Credit) (7 points) We have a two kinds of blocks of size 2×1 or 2×2. We need to fill a big block of size $2 \times n$ with these "small" blocks. **The 2×1 blocks can be arranged either vertically or horizontally**. So for example if we had a block of size 2x2 we could:

1. Fill it with 2 horizontal 2×1 blocks
2. Fill it with 2 vertical 2×1 blocks
3. Fill it with one 2×2 block

Write a racket function that takes n as input outputs the number of ways of filling a $2 \times n$ block as described above.

```
(: cover-floor : Integer → Integer)
(define (cover-floor n)
  (match n
    [0 0]
    [1 1]
    [_ (+ (+ 1 (cover-floor (- n 1)))
          (+ 1 (cover-floor (- n 2)))
          (+ 1 (cover-floor (- n 2))))]))
```

13 . (Extra Credit)

(a) **(5 points)** Two numbers are anagrams of each other if they can both be formed by rearranging the same combination of digits. For example 1246878 and 7481268 are anagrams of each other. Write a racket function that takes two numbers (**Integer** type) as input and outputs true if they are anagrams otherwise false.

(b) **(5 points)** A perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. For instance, 6 has divisors 1, 2 and 3 (excluding itself), and 1 + 2 + 3 = 6, so 6 is a perfect number. The number 28 is also a perfect number since 28 = 1 + 2 + 4 + 7 + 14. Write a racket function `(count-perfect)` that takes a positive integer as input and outputs the number of perfect number less than or equal to your input. For example, `(count-perfect 100)` should output 2, since there are only two perfect numbers 6 and 28 which are less than or equal to 100. Similarly, `(count-perfect 10)` should output 1 since there is only one perfect number 6 which is less than or equal to 10.