Data Structures
CSCI H343, Fall 2024

Midterm

Name: ______________________

This exam has 11 questions, for a total of 100 points.

1. [12 points] What is the big-O time complexity of the following `flood` method in terms of the total number of tiles, represented by n? Provide an argument for your answer that analyzes every statement in the method and how their individual time complexities combine into the total time complexity.

```
void flood(WaterColor color, LinkedList<Coord> flooded_list,
           Tile[][] tiles, Integer board_size) {
    HashSet<Coord> flooded = new HashSet<>(flooded_list);
    for (int i = 0; i != flooded_list.size(); ++i) {
        Coord c = flooded_list.get(i);
        for (Coord n : c.neighbors(board_size)) {
            if (!flooded.contains(n)
                    && tiles[n.getY()][n.getX()].getColor() == color) {
                flooded_list.add(n);
                flooded.add(n);
            }
        }
    }
}
```

Solution:

1. The allocation of the `flooded` HashSet is $O(n)$ **(1 point)**.
2. The outer `for` loop iterates $O(n)$ times. **(1 point)**
   (a) The `get(i)` method call is $O(n)$ **(3 points)**.
   (b) The inner `for` loop iterates at most 4 times and the operations inside it are all $O(1)$, so this loop is $O(1)$ **(3 points)**.
   (c) Thus, the body of the outer `for` loop is $O(n)$ **(1 point)**.

   So the outer `for` loop's time complexity is $O(n^2)$ **(2 points)**.

Thus, the time complexity of `flood` is $O(n^2)$ **(1 point)**.

2. 10 points Complete the following implementation of the Counting Sort algorithm by filling in the blanks.

```
static void counting_sort(int[] A, int[] B, int k) {
   int[] C = new int[k];
   int[] L = new int[k];
   for (int i = 0; i != A.length; ++i) {
          C[___(a)___] += 1;
   }
   L[0] = ___(b)___;
   for (int j = 1; j != k; ++j) {
          L[j] = C[j] + ___(c)__;
   }
   for (int j = A.length - 1; j != -1; --j) {
          B[___(d)___] = A[j];
          L[___(e)___] -= 1;
   }
}
```

Solution: (2 points each)

```
a) A[i]
b) C[0]
c) L[j-1]
d) L[ A[j] ] - 1
e) A[j]
```

3. [8 points] Given the following definition of function `g`, what is the result of `g(2, 1)`? Show your work by listing the arguments and the return value for each call to `g`.

```
public static int g(int m, int n) {
    if (n == 0)
        return m + 2;
    else if (m == 0)
        return g(1, n-1 );
    else
        return g(g(n-1, n), n-1);
}
```

Solution:

The result is 5. **(4 points)**

(1/2 point for each call and return below)

```
g(2,1)
|
|-- g(0, 1)
|   |
|   |-- g(1, 0)
|   |   |
|   |   |- return 1 + 2 = 3
|   |
|   |- return 3
|
|-- g(3, 0)
|   |
|   |- return 3 + 2 = 5
|
|- return 5
```

4. [8 points] Apply the Partition algorithm to the following array, ensuring that all elements less or equal to the pivot element are in lower positions and all elements greater than the pivot are in greater positions. The pivot element starts out as the last element of the array. Write down the initial array and the array after each step (each iteration of the loop), drawing two vertical lines to separate the three partitions (the less-than or equal region, the greater-than region, and the to-do region).

$$[8, 2, 7, 3, 5]$$

Solution:

[|| 8, 2, 7, 3 | 5]	(2 points)
[| 8 | 2, 7, 3 | 5]	(1 point)
[2 | 8 | 7, 3 | 5]	(1 points)
[2 | 8, 7 | 3 | 5]	(1 point)
[2, 3 | 7, 8 || 5]	(1 point)
[2, 3 | 5 | 8, 7]	(2 points)

5. 10 points Show that $4n \log n + 10 \lesssim n^2$ using the definition of asymptotic less-or-equal (aka. big-O).

Solution: Choose $c = 2$ (3 points), $k = 8$ (3 points). (There are other correct answers.)

Demonstration or proof that $4n \log n + 10 < 2n^2$ for $n > 8$: (4 points)

n	$4n \log n + 10$	$2n^2$
1	10	2
2	18	8
4	42	32
8	106	128
16	266	512

6. 10 points For the following `Node` class in a binary tree, fill in the blanks to complete the `previous` method that returns the node that comes before the current node with respect to an inorder traversal, if there is one, and `null` if there is none.

```
class Node {
    T data;
    Node left, right, parent;
    public Node previous() {
        if (left == null) {
            return ___(a)___;
        } else {
            return ___(b)___;
        }
    }
    public Node last() {
        if (right == null)
            return this;
        else
            return ___(c)__;
    }
    public Node prevAncestor() {
        Node q = this;
        Node p = q.parent;
        while (p != null && ___(d)___) {
            q = p;
            p = ___(e)___;
        }
        return p;
    }
}
```

Solution: Rubric: 2 points each

```
a) prevAncestor()
b) left.last()
c) right.last()
d) p.left == q
e) p.parent
```

7. [10 points] Given the following definitions of the `sum` and `rev_app` functions, prove the `sum_rev_app` theorem specified below. The proof can be carefully written in English or in the Deduce language.

```
function sum(List<Nat>) -> Nat {
  sum(empty) = 0
  sum(node(x, xs)) = x + sum(xs)
}

function rev_app<T>(List<T>, List<T>) -> List<T> {
  rev_app(empty, ys) = ys
  rev_app(node(x, xs), ys) = rev_app(xs, node(x, ys))
}

theorem sum_rev_app: all xs : List<Nat>. all ys:List<Nat>.
  sum(rev_app(xs, ys)) = sum(xs) + sum(ys)
```

Solution: It's OK for the student's proof to gloss over obvious rules of arithmetic such as associativity and commutativity of addition.

```
theorem sum_rev_app: all xs : List<Nat>. all ys : List<Nat>.
  sum(rev_app(xs, ys)) = sum(xs) + sum(ys)
proof
  induction List<Nat>
  case empty {
    arbitrary ys:List<Nat>
    suffices sum(ys) = 0 + sum(ys)
      by definition {rev_app, sum}
    symmetric zero_add[sum(ys)]
  }
  case node(x, xs') assume IH {
    arbitrary ys:List<Nat>
    suffices sum(rev_app(xs', node(x, ys))) = (x + sum(xs')) + sum(ys)
      by definition {sum, rev_app}
    equations
          sum(rev_app(xs', node(x, ys)))
        = sum(xs') + sum(node(x, ys))    by IH[node(x,ys)]
    ... = sum(xs') + (x + sum(ys))       by definition sum
    ... = (sum(xs')+x) + sum(ys) by symmetric add_assoc[sum(xs')][x,sum(ys)]
    ... = (x + sum(xs')) + sum(ys)       by rewrite add_commute[sum(xs')][x]

  }
end
```

8. 8 points Which of the following are valid tests for the `find_first_true` function? A valid test is one that satisfies the preconditions of `find_first_true` and that checks for the correct result. For each of the five tests below, write "valid" or "invalid" next to it. Recall the specification for `find_first_true`:

Specification: The `find_first_true(A, begin, end)` function returns the smallest index `i` in the half-open range specified by `begin` and `end` such that `A[i]` is true. If there are no true elements in the range, then `find_first_true` returns the `end` position. The caller of `find_first_true` is required to provide a valid half-open range for array `A`, which means `begin <= end`, `0 <= begin`, `begin <= A.length`, `0 <= end`, and `end <= A.length`.

1.
```
boolean[] A = {true, false};
assertEquals(0, Search.find_first_true(A, 1, 0));
```
2.
```
boolean[] A = {true, false, true};
assertEquals(2, Search.find_first_true(A, 1, 3));
```
3.
```
boolean[] A = {false, false, false};
assertEquals(2, Search.find_first_true(A,0,2));
```
4.
```
boolean[] A = {true, true, false};
assertEquals(1, Search.find_first_true(A, 0, 3));
```

Solution: (2 points each)

1. invalid (`begin` is greater than `end`)
2. valid
3. valid
4. invalid (the return value should be `2`, not `0`)

9. 10 points Implement a generic version of the Quicksort Algorithm that sorts a half-open range of elements, given by a pair of iterators. Recall the following definition of the `Iterator` interface. You do not need to implement the `partition` helper function.

```
interface Iterator<T> {
    T get();
    void set(T e);
    void advance();
    boolean equals(Iterator<T> other);
    Iterator<T> clone();
}
static <E extends Comparable<? super E>>
Iterator<E> partition(Iterator<E> begin, Iterator<E> end);

static <E extends Comparable<? super E>>
void quicksort(Iterator<E> begin, Iterator<E> end) {
```

Solution:

```
public static <E extends Comparable<? super E>>
void quicksort(Iterator<E> begin, Iterator<E> end) {
    if (! begin.equals(end)) {    // (2 points)
        Iterator<E> pivot = partition(begin, end); // (2 points)
        quicksort(begin, pivot); // (2 points)
        pivot.advance();         // (2 points)
        quicksort(pivot, end);   // (2 points)
    }
}
```

10. [8 points] What are the big-O time complexities for the `get` and `contains` methods of the Java `LinkedList` class?

Solution: (2 points each)

1. get: $O(n)$
2. contains: $O(n)$

What are the big-O time complexities for the `get` and `contains` methods of the Java `ArrayList` class?

Solution: (2 points each)

1. get: $O(1)$
2. contains: $O(n)$

11. [6 points] What is the big-O time complexity of the following `insertion_sort` function? Explain your answer in detail, giving the big-O for the `insert` and `isort` helper functions. You may assume that the `<=` operator is $O(1)$.

```
function insert(List<Nat>,Nat) -> List<Nat> {
  insert(empty, x) = node(x, empty)
  insert(node(y, next), x) =
    if x <= y then
      node(x, node(y, next))
    else
      node(y, insert(next, x))
}
function isort(List<Nat>, List<Nat>) -> List<Nat> {
  isort(empty, ys) = ys
  isort(node(x, xs), ys) = isort(xs, insert(ys, x))
}
define insertion_sort : fn List<Nat> -> List<Nat>
    = fun xs{ isort(xs, empty) }
```

Solution: The `insert` function is $O(n)$ because it traverses its input list and does $O(1)$ work per node in the list. **(2 points)**

The `isort` function is $O(n^2)$ because it traverses the list `xs` and for each element it invokes `insert`, so we have n iterations **(1 point)** times $O(n)$ **(1 point)** which is $O(n^2)$ **(1 point)**.

The `insertion_sort` function is $O(n^2)$ because it calls `isort`. **(1 point)**