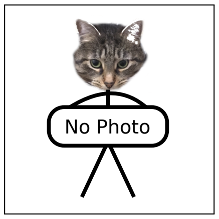
Midterm Exam

CSE 250 — Fall 2021 — **Section**

Instructor: Oliver Kennedy

Name:
UBIT:
Person #:
Seat Number: Spare 1



Academic Integrity

My signature on this cover sheet indicates that I agree to abide by the academic integrity policies of this course, the department, and university, and that this exam is my own work.

Name: ____________________ Date: ____________________

Instructions

Check that your name and UBIT are correct on this cover sheet and the pages of this exam. If you have the wrong exam, *notify an instructor or TA immediately!*

Answer each question on this exam to the best of your ability. You may make notes or perform calculations in the margins or any blank area on the bottom or margins of exam pages, on the designated scratch pages, or on the back of this cover sheet. If you mis-mark an answer and need to correct it, *draw a line through the mis-marked answer and circle the corrected answer.*

Questions vary in difficulty. *Do not get stuck on one question.* When you are finished, check to ensure that you have answered all questions, then turn in the entire exam (including all scrap pages used) to a TA or instructor.

Part	**Topic**	**Max Points**	**Earned Points**
1	Scala Types	5	
2	Scala Objects	10	
3	Scala Mutability	10	
4	Asymptotic Analysis	20	
5	The List Adt	15	
6	Linked Lists	10	
7	Stacks And Queues	5	
8	Graphs	15	
9	Graph Runtime	10	
Total		100	

Part 1: Scala Types (5 points total)

Consider the following scala code:

```scala
val x = 7.0f * 8 + 9.2
```

Question 1 (5 points)

What is the type of x? (pick one)

- ☐ Integer
- ☐ Long
- ☐ Float
- x **Double**

Part 2: Scala Objects (10 points total)

Consider the following definition:

```
trait Foo
{
  def apply(): Unit = println("Hello World!")
}
```

Question 2 (5 points)

Foo may be instantiated (i.e., writing new Foo in the scala interpreter will not throw an error).

Circle One: True / [[False]]

Question 3 (5 points)

A class may extend Foo

Circle One: [[True]] / False

Part 3: Scala Mutability (10 points total)

Consider the variable `seq` defined and manipulated as follows:

```scala
var seq = scala.immutable.Seq('A', 'B', 'C')
myFunctionThatDoesThings(seq)
```

Question 4 (5 points)

After the initialization process above (and not knowing anything about how `myFunctionThatDoesThings` is implemented), it is 100% safe to assume that `seq(2)` returns `'C'`

Circle One: [[True]] / False

Question 5 (5 points)

After the initialization process above, the following operation is valid scala. In other words, if the following line were appended to the line above, the compiler would not indicate an error.

```scala
seq = scala.immutable.Seq('Q', 'R', 'S')
```

Circle One: [[True]] / False

Part 4: Asymptotic Analysis (20 points total)

Consider the following formulas:

$$f(n) = 5n^2 \log^2(n) + 8n^2 + 2^{\log(20n)}$$
$$g(n) = 2n\log(n) + 3n + 9\log(n \cdot 2^n)$$

Question 6 (10 points)

Provide the simplified tight lower-bound for $f(n)$ with the appropriate asymptotic choice (O, Omega, Theta).

The lower bound operation is Ω. A "tight" bound means that we can't get a lower complexity class. Since the curve of the equation is smooth (i.e., there is no "case" style behavior), this is always the dominant term in the expression.
A sufficient answer to this question is $\Omega(n^2\log^2(n))$
To prove to yourself that $n^2\log^2(n)$ is the dominant term, consider:
First, for sufficiently large n, is

$$n^2\log^2(n) \geq n^2$$
$$\log^2(n) \geq 1$$

True!
Second, for sufficiently large n, is

$$n^2\log^2(n) \geq 2^{\log(20n)}$$
$$n^2\log^2(n) \geq 20n$$
$$n\log^2(n) \geq 20$$

True!

Question 7 (10 points)

Provide the simplified tight upper-bound for $g(n)$ with the appropriate asymptotic choice (O, Omega, Theta).

The dominant term here is not entirely obvious due to the term at the end. By the log rules:

$$\log(n \cdot 2^n) = \log(n) + \log(2^n) = \log(n) + n$$

So, the dominant term is $n\log(n)$
The upper bound is O, and its tight value is the dominant term.
A sufficient answer to this question is $O(n\log(n))$

Part 5: The List ADT (15 points total)

Recall `FixedArrayListV3`, the final variant of the `ListADT` abstract data type we implemented in class. (We later called this the `ArrayBuffer`, or resizable array). Assume that the variable `arr` is instantiated as follows

```
val n = Random.nextInt(100000)
val arr = new FixedArrayListV3[Int]()
arr.reserve(n)
for(i ← 0 until n){ arr.insert(idx = i, elem = 0) }
```

For each of the following snippets of code, state its runtime complexity in terms of `n` (and/or `m` if applicable).

Question 8 (7 points)

```
arr.insert(idx = k, elem = 42)
```

$\Theta(n-k)$, $\Theta(n)$, $O(n-k)$, or $O(n)$ would all be acceptable answers.

Question 9 (8 points)

```
val m = Random.nextInt(1000000)
for(i ← 0 until m) { arr.insert(idx = arr.length, elem = m) }
```

$\Theta(m)$ or $O(m)$ would both be acceptable answers.
Notably, any answer involving a multiplicative n term (e.g., $O(m \cdot n)$) would **not** be a correct answer, since append (i.e., inserting at the end) always has an *amortized* $O(1)$ runtime. $O(m+n)$ is technically correct as well.

Part 6: Linked Lists (10 points total)

Recall the `LinkedListBuffer` implementation of a linked list that you're working on for PA1. Consider an instance of `LinkedListBuffer[String]` with the following state:

`_value ="Maine Coon"`		`_value ="Ocicat"`		`_value ="Longhair"`		`_value ="Birman"`	
`_prev =-1`	`_next =2`	`_prev =2`	`_next =3`	`_prev =0`	`_next =1`	`_prev =1`	`_next =-1`

```
_head =0
_tail =3
_numStored =4
```

Question 10 (10 points)

What is the sequence that the buffer encodes (i.e., what sequence of values does `buffer.iterator` produce)?

Starting at the index referenced by the `_head` variable, we iterate following the `_next` variables, producing the sequence:

1. Main Coon
2. Longhair
3. Ocicat
4. Birman

Part 7: Stacks and Queues (5 points total)

Consider the following code:

```
 1  val seq1 = Seq(17, 73, 65, 0)
 2  val seq2 = Seq(45, 1, 14, 48)
 3  val queue = scala.collection.mutable.Queue()
 4
 5  for(i ← seq1){ queue.enqueue(i) }
 6  for(i ← 0 until 3){ queue.dequeue() }
 7  for(i ← seq2){ queue.enqueue(i) }
 8  while(!queue.isEmpty){
 9    println(queue.dequeue())
10  }
```

Question 11 (5 points)

What does the above code print?

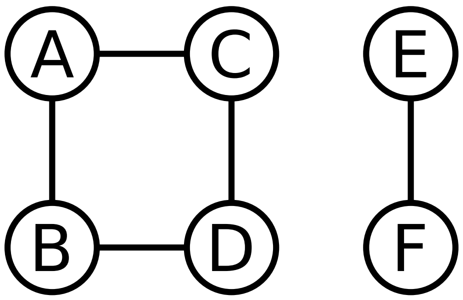
- Line 5, enqueue all of `seq1`: **queue** = `17, 73, 65, 0`
- Line 6, dequeue 3 times: **queue** = `0`
- Line 7, enqueue all of `seq2`: **queue** = `0, 45, 1, 14, 48`
- Lines 8-10, print the queue in order

The final output will be:

```
0
45
1
14
48
```

Part 8: Graphs (15 points total)

Consider the following graph `G`:



Question 12 (5 points)

How many vertices are in the largest connected component?

The two connected components are A, B, C, D and E, F, so 4 in the bigger one.

Question 13 (10 points)

List the vertices visited by `DFSOne(G, A)` **in the order in which they are visited**. Assume that iteration over incidence lists happens in ascending alphabetical order of the opposite edge. For example, starting from a hypothetical vertex X, the edge (X, Y) would be traversed before the edge (X, Z).

A sufficient answer would be A, B, D, C

The detailed execution trace is:

1. DFS visits A, calling DFSOne on it
2. DFSOne moves from A to B
3. DFSOne checks the B to A edge, but the edge is already marked visited.
4. DFSOne moves from B to D
5. DFSOne checks the D to B edge, but the edge is already marked visited
6. DFSOne moves from D to C
7. DFSOne checks the C to A edge. The edge is unvisited, but A has been visited, so the edge is marked as a back-edge.
8. DFSOne checks the C to D edge, but the edge is already marked visited
9. DFSOne is done visiting C's edges and backtracks to D
10. DFSOne is done visiting D's edges and backtracks to B
11. DFSOne is done visiting B's edges and backtracks to A
12. DFSOne checks the A to C edge, but the edge is already marked visited
13. DFSOne is done visiting A's edges and backtracks to the outer DFS call
14. DFS checks B, but it is already visited.
15. DFS checks C, but it is already visited.
16. DFS checks D, but it is already visited.
17. DFS visits E, calling DFSOne on it
18. DFSOne moves from E to F
19. DFSOne checks the F to E edge, but the edge is already marked visited.
20. DFSOne is done visiting F's edges and backtracks to E
21. DFSOne is done visiting E's edges and backtracks to the outer DFS call
22. DFS checks F, but it is already visited.

Part 9: Graph Runtime (10 points total)

Question 14 (10 points)

As we discussed in class, the runtime of BFS is $O(|V| + |E|)$. However, this runtime assumes that the graph is stored in an *adjacency list* structure. What is the runtime if we instead use an *edge list*. Recall that an *edge list* is an *adjacency list* without the list of incident edges (or in/out edges) for each vertex.

The key difference between these two data structures is that the edge list requires $O(|E|)$ time to enumerate the edges incident on a specific vertex $v \in V$, while the adjacency list can do so in $O(deg(v))$. BFSOne iterates through all of the adjacent edges once per node it visits, so its runtime *per node visited* is $O(deg(v))$. Since it visits each node exactly once, the total runtime is $O(\sum_{v \in V} deg(v)) = O(|E|)$. If the cost of enumerating incident edges goes up to $O(|E|)$, we get a runtime of $O(\sum_{v \in V} |E|) = O(|V||E|)$, which becomes the dominant term in the runtime.

$O(|V| \cdot |E|)$