COS 126 Written Exam 2 Fall 2018

There are nine questions on this exam, weighted as indicated. The questions are ordered corresponding to the lectures, *not in order of difficulty*. If a question seems difficult to you, skip it and come back to it. You will have 50 minutes to complete the exam.

This exam is preprocessed by computer. You answer questions by filling in circles *completely* with a dark pencil. If you change your mind, you must erase *completely* and fill in another circle!

Do this ● not this  or this  or this  .

Resources. You may reference your optional two-sided 8.5-by-11 handwritten "cheat sheet" during this exam. You may not use the textbook, your notes, or any electronic devices. You may not communicate with anyone except the course staff during this exam.

Discussing this exam. Due to travel for extracurriculars and sports, some of your peers will take this exam later. Do not discuss its contents with anyone who has not taken it.

This page. Do not remove this exam from the exam room. Fill in this page now, but do not start the exam until you are told.

Name	
NetID	
Precept	
Exam Room	

"I pledge my honor that I have not violated the Honor Code during this examination."

_____

_____

[*copy the pledge here*]

_____

[*signature*]

Q1. Data Type Terminology (9 points).

Consider the following Java code.

```
 1   public class SimpleStack
 2   {
 3      private String[] a;
 4      private int N = 0;
 5      public SimpleStack(int max)
 6      { a = new String[max]; }
 7      public void push(String item)
 8      { a[N++] = item; }
 9      public String pop()
10      { return a[--N]; }
11      public static void main(String[] args)
12      {
13         SimpleStack stack = new SimpleStack(10);
14         stack.push("test");
15         String s = stack.pop();
16         StdOut.println(s);
17      }
18   }
```

Fill in one or more squares in each row that correspond to line numbers that fit the description.

	1	3	4	5-6	7-8	11-17	13	14	15
creates an object	☐	☐	☐	☐	☐	☐	☐	☐	☐
invokes a method	☐	☐	☐	☐	☐	☐	☐	☐	☐
invokes a construtor	☐	☐	☐	☐	☐	☐	☐	☐	☐
names a class	☐	☐	☐	☐	☐	☐	☐	☐	☐
declares and initializes a local variable	☐	☐	☐	☐	☐	☐	☐	☐	☐
implements an instance method	☐	☐	☐	☐	☐	☐	☐	☐	☐
declares and initializes an instance variable	☐	☐	☐	☐	☐	☐	☐	☐	☐
defines a constructor	☐	☐	☐	☐	☐	☐	☐	☐	☐
implements a static method	☐	☐	☐	☐	☐	☐	☐	☐	☐

Q2. Performance (6 points).

On each row, fill in the one circle that gives the best hypothesis for the order of growth of the running time of a program having the given observed running times.

observed running time				*order-of-growth hypothesis*			
1 million inputs	2 million inputs	4 million inputs	12 million inputs	linear	quadratic	cubic	none of these
	.25 seconds	2 seconds	about 1 minute	○	○	○	○
.5 seconds		2 seconds	6 seconds	○	○	○	○
1 second	4 seconds	32 seconds	4-5 hours	○	○	○	○
15 minutes		6 hours	2-3 weeks	○	○	○	○
30 minutes	4 hours	1-2 days	1-2 months	○	○	○	○
15 minutes		6 hours	2-3 days	○	○	○	○

Q3. Searching and Sorting (8 points).

To the right of each option, fill in the circle corresponding to the one-word characterization that best describes the order of growth of the ***worst-case*** running time.

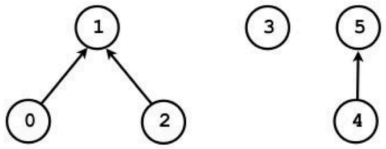
	logarithmic ($\log N$)	linear (N)	linearithmic ($N \log N$)	quadratic (N^2)
mergesort	○	○	○	○
last merge in mergesort	○	○	○	○
binary search	○	○	○	○
BST construction	○	○	○	○
search in a BST	○	○	○	○
insertion sort	○	○	○	○
sequential search	○	○	○	○
bubble sort	○	○	○	○

Q4. Data Types and Linked Structures (5 points).

Consider the following code, which builds general structures known as forests, where roots have null links and every other node points to its parent.

```
public class Forest
{
   private Node[] links;
   private class Node
    {  private Node next;  }
   public Forest(int N)
   {
      links = new Node[N];
      for (int i = 0; i < N; i++)
         links[i] = new Node();
   }
   private Node root(int i)
   {
      Node x = links[i];
      while (x.next != null) x = x.next;
      return x;
   }
   public void merge(int i, int j)
    {  root(i).next = root(j); }
   public boolean isMerged(int i, int j)
    {  return root(i) == root(j); }
}
```

```
Forest t = new Forest(6);
t.merge(0, 1);
t.merge(2, 1);
t.merge(4, 5);
```



To begin to understand how this code works, verify that the client code to the right of the code above produces the forest shown below it. In this diagram, each node is labeled with the index in the links array that has a reference to it. You may find it useful to draw a diagram like this on scratch paper to answer this question (but we will not look at your diagram).

```
Forest t = new Forest(8);
t.merge(0, 3);
t.merge(1, 2);
t.merge(1, 4);
t.merge(5, 6);
t.merge(3, 4);
t.merge(7, 5);
```

Suppose that the client code at right is executed. In each row in the table below, fill in the circle corresponding to the value returned by the given call to `t.isMerged()`.

	true	**false**
`t.isMerged(0, 3);`	◯	◯
`t.isMerged(2, 3);`	◯	◯
`t.isMerged(0, 7);`	◯	◯
`t.isMerged(1, 3);`	◯	◯
`t.isMerged(4, 5);`	◯	◯

Q5. BSTs (10 points).

A. Suppose that we create a BST by inserting two-character strings into an initially empty tree. In the blank to the right of each of the following insertion orders write the height of the tree produced (max number of links on any path from the root to a leaf node) when keys are inserted in that order into an initially empty tree. The first answer is provided for you.

	3	4	5	6	7
do, fa, la, mi, re, so, ti	○	○	○	○	●
do, re, mi, fa, so, la, ti	○	○	○	○	○
ti, do, so, fa, re, la, mi	○	○	○	○	○
ti, do, fa, so, la, re, mi	○	○	○	○	○
fa, la, re, ti, so, mi, do	○	○	○	○	○
mi, so, fa, ti, re, la, do	○	○	○	○	○

B. Suppose that you are searching for the key ho in a binary search tree built with two-character string keys. In the following table, fill in one circle in each row to indicate whether it is possible or impossible for keys given to be examined in the order given during the search.

	possible	*impossible*
do, ti, so, fa, mi, re, la	○	○
do, la, ti, fa, so, re, mi	○	○
do, fa, la, mi, re, so, ti	○	○
do, re, mi, fa, so, la, ti	○	○
fa, la, re, ti, so, mi, do	○	○

Q6. Regular Expressions (8 points)

Let L = { babba, babaa, bbbba, baaba, baaaa }. For each of the regular expressions at left in the table below, fill in the circle in the A, B, C, or D column to indicate whether the RE

NONE Matches no strings in L.

LESS Matches at least one string in L (but not all) and no strings that are not in L.

POOR Matches at least one string in L (but not all) and at least one other string.

MORE Matches all strings in L and at least one other string.

EXACT Matches all strings in L and no strings that are not in L.

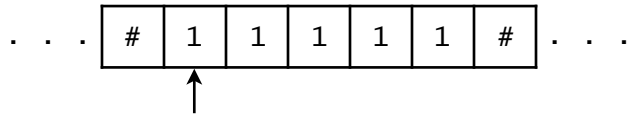
Fill in exactly one circle in each row.

	NONE	LESS	POOR	MORE	EXACT
a(a|b)*ab(a|b)*	◯	◯	◯	◯	◯
ba(a|b)aa	◯	◯	◯	◯	◯
b*a*b*a	◯	◯	◯	◯	◯
bbbba|(ba(a|b)(a|b)a)	◯	◯	◯	◯	◯
a*b*aa*b*ba*a*ca*b*b*a*a*b*	◯	◯	◯	◯	◯
b(a|b)(a|b)(a|b)a	◯	◯	◯	◯	◯
b((a(a|b)(a|b))|bbb)a	◯	◯	◯	◯	◯
baa(a|b)a*	◯	◯	◯	◯	◯

Q7. Turing Machines (7 points).

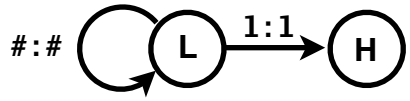
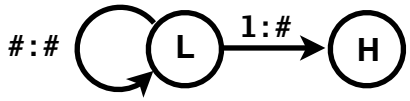
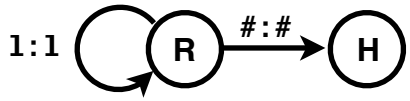
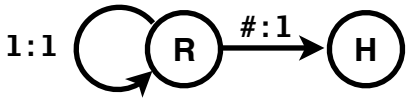
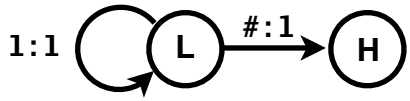
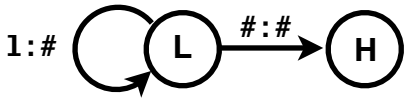
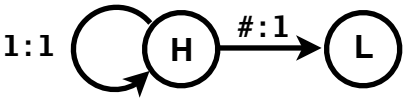
A *unary Turing machine* has just one tape symbol (1) in addition to the marker symbol #. Otherwise, it has all the properties of any other Turing machine.

- The initial tape consists of a string of 1s with infinitely many # symbols on both sides, the initial head location is the leftmost bit of the string of 1s, and the *value* of the tape at any point is the number of 1s on the tape. For example this diagram depicts a tape of value 5.



- The initial state of each machine is the leftmost in the diagram, and the next state is determined by following the arrow whose label before the colon matches the symbol at the head location on the tape. The label after the colon is written on the tape. On entering the new state, the head location moves left (L) or right (R) as indicated, or halts (H). Then the process continues.

In the table below, fill in exactly one circle in each row to describe the computation performed by the given unary Turing machine.

	increments value	*decrements value*	*leaves value unchanged*	*does not halt*
	○	○	○	○
	○	○	○	○
	○	○	○	○
	○	○	○	○
	○	○	○	○
	○	○	○	○
	○	○	○	○

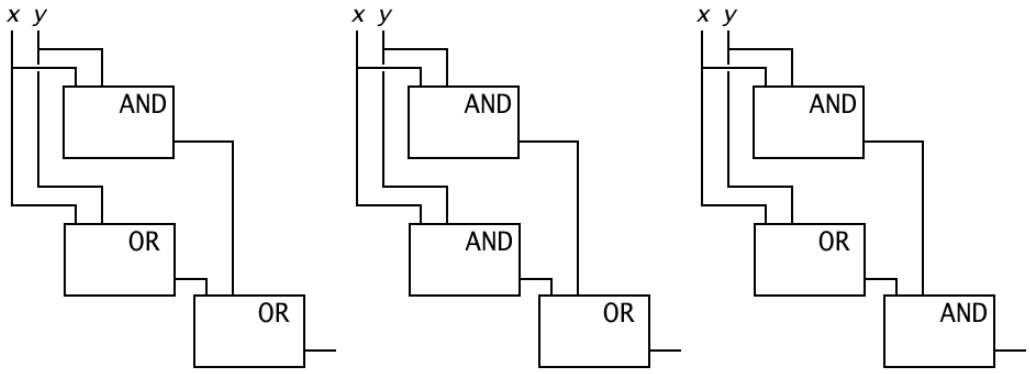
Q8. Intractability (9 points).

Fill one circle in each row to indicate whether the statement is **TRUE** or **FALSE**. If the statement is not known to be either true or false, fill in the circle in the ? column. You must fill in exactly one circle in each column.

	TRUE	FALSE	?
If P ≠ NP, there is no polynomial-time algorithm for integer linear programming.	◯	◯	◯
If P = NP, every problem in P is NP-complete.	◯	◯	◯
If P = NP there is a polynomial-time algorithm for factoring.	◯	◯	◯
Factoring polynomial-time reduces to satisfiability.	◯	◯	◯
No problem is in both P and NP.	◯	◯	◯
There exists a deterministic TM that can solve every problem in NP.	◯	◯	◯
A Universal Turing Machine can simulate the operation of any Turing machine, including itself, in polynomial time.	◯	◯	◯
If P ≠ NP, there is no polynomial-time algorithm for factoring.	◯	◯	◯
There may exist an exponential time solution for the Halting Problem.	◯	◯	◯

Q9. Combinational Circuits/Boolean Logic (8 points).

Consider the following three circuits:



A **B** **C**

Fill in one more squares in each row to indicate which circuit computes the given function. If none of them do so, fill in the square in the NONE column.

	A	B	C	NONE
x OR y	☐	☐	☐	☐
x AND y	☐	☐	☐	☐
x XOR y	☐	☐	☐	☐
(x OR y) OR x	☐	☐	☐	☐
(x AND y) OR (x AND y)	☐	☐	☐	☐
(x OR y) OR (x AND y)	☐	☐	☐	☐
(((x OR y) AND x) AND y	☐	☐	☐	☐
(((x AND y) OR x) OR y	☐	☐	☐	☐