COS 126	Princeton University	Fall 2022

Written Exam 2

This exam has 10 questions worth a total of 50 points. You have 70 minutes.

Instructions. This exam is preprocessed by computer. Write neatly, legibly, and darkly. Put all answers (and nothing else) inside the designated spaces. *Fill in* bubbles and checkboxes completely: ● and ■. To change an answer, erase it completely and redo.

Resources. The exam is closed book, except that you are allowed to use a one page reference sheet (8.5-by-11 paper, both sides, in your own handwriting). No electronic devices are permitted.

Honor Code. This exam is governed by Princeton's Honor Code. Discussing the contents of this exam before solutions have been posted is a violation of the Honor Code.

Please complete the following information now.

Name:

NetID:

Exam room: ○ McCosh 10 ○ McCosh 50 ○ McCosh 62 ○ Other

Precept:

P01	P02	P03	P04	P05	P06	P07	P08	P08A
○	○	○	○	○	○	○	○	○

P10	P11	P12	P12A	P13	P14	P15
○	○	○	○	○	○	○

"I pledge my honor that I will not violate the Honor Code during this examination."

Signature

1. **Initialization. (1 point)**

   On the front of this exam, in the designated spaces, write your name and NetID (not email alias); mark the room in which you are taking the exam and your precept number; and write and sign the Honor Code pledge.

2. **Java OOP properties. (7 points)**

   Which of the following are properties of object-oriented programming in Java?

   Mark each statement as either true or false by filling in the appropriate bubble.

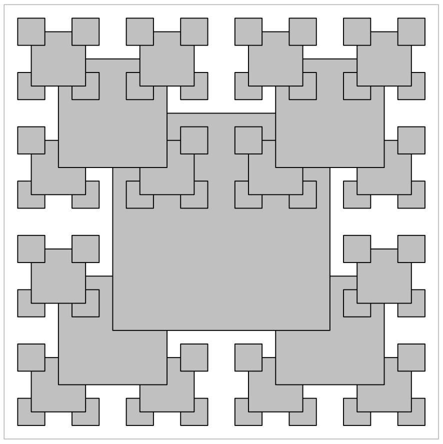
true	*false*	
●	○	Java has two kinds of types—*primitive types* and *reference types.*
○	○	Every reference variable has a type (such as `String` or `Perceptron`) that is known at compile time.
○	○	Every class has *exactly one* constructor.
○	○	Programmers typically declare instance variables to be `public` in order to make the data type easier to test, debug, and maintain.
○	○	If you do not explicitly initialize an instance variable, it is initialized automatically to a default value (such as `0`, `0.0`, or `null`).
○	○	If `a` and `b` refer to two `String` objects, then `a == b` checks whether they correspond to the same sequence of characters.
○	○	The `String` data type is *immutable*: it is not possible to change the value of a `String` object by calling one of its public instance methods.
○	○	If `x` is a reference variable, then, when evaluating the expression `"x = " + x`, Java automatically calls the `toString()` method for `x`, then concatenates the two strings.

3. **Recursive graphics. (4 points)**

   Design a recursive function with the signature

```
public static void draw(int n, double x, double y, double length)
```

   so that the call `draw(4, 0.5, 0.5, 0.5)` produces the following drawing:



   These are the six statements in the function body, but not necessarily in the order given:

```
1.  if (n == 0) return;
2.  drawShadedSquare(x, y, length);
3.  draw(n-1, x - length/2, y + length/2, length/2);   // upper left
4.  draw(n-1, x + length/2, y + length/2, length/2);   // upper right
5.  draw(n-1, x - length/2, y - length/2, length/2);   // lower left
6.  draw(n-1, x + length/2, y - length/2, length/2);   // lower right
```

   The helper function `drawShadedSquare()` draws a gray square of the specified side length, outlined in black, and centered at (x, y).

   Which of the following must be true for *any* possible ordering of the six statements that produces the drawing above?

   Mark all that apply.

true	*false*	
○	○	Statement 1 must appear *first*.
○	○	Statement 2 must appear *last*.
○	○	Statement 3 must appear *before* statement 4.
○	○	Both statements 5 and 6 must appear *before* statement 2.

4. **Data-type design and debugging. (6 points)**

   Recall the `Vector` data type from lecture, precept, and the textbook. Consider the following partial implementation and client:

```
public class Vector {
    private double[] coords;  // coords[i] = ith coordinate of vector
    private int n;            // length of vector

    // constructs a new vector using given coordinates
    public Vector(double[] coordinates) {
        // SEE IMPLEMENTATIONS ON FACING PAGE
    }

    // returns the dot product of the two vectors
    public double dot(Vector that) {
        double sum = 0.0;
        for (int i = 0; i < n; i++) {
            sum += this.coords[i] * that.coords[i];
        }
        return sum;
    }

    // a test client
    public static void main(String[] args) {
        double[] x = { 3.0, 2.0, 1.0 };
        Vector a = new Vector(x);
        x[0] = 1.0;
        Vector b = new Vector(x);
        x[0] = 2.0;
        System.out.println(a.dot(b));
    }
}
```

Suppose that you run the `main()` on the facing page with each of the constructor implementations on the left. What will be printed to standard output?

For each implementation on the left, write the letter of the best-matching description on the right. You may use each letter once, more than once, or not at all.

☐

```
public Vector(double[] coordinates) {
    n = coordinates.length;
    coords = coordinates;
}
```

☐

```
public Vector(double[] coordinates) {
    this.n = coordinates.length;
    this.coords = new double[n];
    for (int i = 0; i < n; i++)
        coords[i] = coordinates[i];
}
```

☐

```
public Vector(double[] coordinates) {
    int n = coordinates.length;
    double[] coords = new double[n];
    for (int i = 0; i < n; i++)
        coords[i] = coordinates[i];
}
```

A. `0.0`

B. `1.0`

C. `2.0`

D. `6.0`

E. `8.0`

F. `9.0`

G. `13.0`

H. `14.0`

I. *run-time exception*

5. **TOY. (6 points)**

For each description on the left, write the letter of the best-matching power of 2 on the right. You may use each letter once, more than once, or not at all.

☐	Number of 1s in the binary representation of the decimal integer 27.	**A.** 2^0 (1)
		B. 2^1 (2)
☐	Number of 1s in the binary representation of −1. *Assume 16-bit two's complement integer.*	**C.** 2^2 (4)
		D. 2^3 (8)
☐	Number of distinct integers representable by a TOY register.	**E.** 2^4 (16)
		F. 2^5 (32)
☐	Number of *bytes* of main memory in TOY (including FF).	**G.** 2^6 (64)
		H. 2^7 (128)
☐	Value in `R[3]` after executing the TOY code, starting from `10`:	**I.** 2^8 (256)
		J. 2^9 (512)
		K. 2^{10} (1,024)
☐	Number of times the instruction `92FF` is executed when running the following TOY code, starting from `10`:	**L.** 2^{11} (2,048)
		M. 2^{12} (4,096)
		N. 2^{13} (8,192)
		O. 2^{14} (16,384)
		P. 2^{15} (32,768)
		Q. 2^{16} (65,536)

Code for the fifth item:

```
10: 7113
11: 720D
12: 1312
13: 0000
```

Code for the sixth item:

```
10: 7101    R[1] <- 0001
11: 720F    R[2] <- 000F
12: 92FF    print R[2]
13: 6221    R[2] <- R[2] >> 1
14: D212    if (R[2] > 0) goto 12
15: 0000    halt
```

```
                        TOY REFERENCE CARD



INSTRUCTION FORMATS

              | . . . . | . . . . | . . . . | . . . .|
  Format RR:  | opcode  |    d    |    s    |    t    |  (1-6, A-B)
  Format A:   | opcode  |    d    |       addr        |  (7-9, C-F)


ARITHMETIC and LOGICAL operations
    1: add              R[d] <- R[s] +  R[t]
    2: subtract         R[d] <- R[s] -  R[t]
    3: and              R[d] <- R[s] &  R[t]
    4: xor              R[d] <- R[s] ^  R[t]
    5: shift left       R[d] <- R[s] << R[t]
    6: shift right      R[d] <- R[s] >> R[t]

TRANSFER between registers and memory
    7: load address     R[d] <- addr
    8: load             R[d] <- M[addr]
    9: store            M[addr] <- R[d]
    A: load indirect    R[d] <- M[R[t]]
    B: store indirect   M[R[t]] <- R[d]

CONTROL
    0: halt             halt
    C: branch zero      if (R[d] == 0) PC <- addr
    D: branch positive  if (R[d] >  0) PC <- addr
    E: jump register    PC <- R[d]
    F: jump and link    R[d] <- PC; PC <- addr


Register 0 always reads 0.
Loads from M[FF] come from stdin.
Stores to  M[FF] go to stdout.

16-bit registers (using two's complement arithmetic)
16-bit memory locations
 8-bit program counter
```

6. **Linked lists. (4 points)**

Suppose that the `Node` data type is defined as

```
private class Node {
   private int item;
   private Node next;
}
```

and that `first` is a variable of type `Node` that refers to the first node in a *circular* linked list. Assume the circular linked list contains at least two nodes.

For each code fragment on the left, write the letter of the best-matching description on the right. You may use each letter once, more than once, or not at all.

```
for (Node x = first; x != first; x = x.next)
   StdOut.println(x.item);
```

```
for (Node x = first; x != null; x = x.next)
   StdOut.println(x.item);
```

```
Node x = first;
while (x.next != first) {
   StdOut.println(x.item);
   x = x.next;
}
```

```
Node x = first;
do {
   StdOut.println(x.item);
   x = x.next;
} while (x != first);
```

A. prints all items once

B. prints all items once except the first one

C. prints all items once except the last one

D. prints only the first item

E. prints only the last item

F. prints nothing

G. infinite printing loop

7. **Algorithms, data structures, and performance. (5 points)**

Suppose that the following code fragment is used to initialize `array`, `stack`, `queue`, and `bst`:

```
int[] array = { 2, 1, 1, 3, 1, 2, 1, 1, 1 };
int n = array.length;
Stack<Integer> stack = new Stack<Integer>();
Queue<Integer> queue = new Queue<Integer>();
ST<Integer, Integer> st = new ST<Integer, Integer>();

for (int i = 0; i < n; i++) {
   stack.push(array[i]);
   queue.enqueue(array[i]);
   st.put(array[i], i);
}
```

What is the *order of growth* of the running time of the above `for` loop in the *worst case* as a function of the array length n? Recall that `ST` is implemented using a *balanced* BST.

○ 1 ○ $\log n$ ○ n ○ $n \log n$ ○ n^2 ○ 2^n

For each code fragment on the left, write the letter of the best-matching output on the right. You may use each letter once, more than once, or not at all.

☐
```
while (!stack.isEmpty())
   StdOut.print(stack.pop() + " ");
```

☐
```
while (!queue.isEmpty())
   StdOut.print(queue.dequeue() + " ");
```

☐
```
for (int key : st.keys())
   StdOut.print(key + " ");
```

☐
```
for (int i = 0; i < n; i++)
    StdOut.print(st.get(array[i]) + " ");
```

A. 0 1 2 3 4 5 6 7 8

B. 1 1 1 1 1 1 2 2 3

C. 1 1 1 2 1 3 1 1 2

D. 2 1 1 3 1 2 1 1 1

E. 5 8 8 3 8 5 8 8 8

F. 1 2 3

G. 1 2 6

H. 6 2 1

I. 8 5 3

8. **Regular expressions and DFAs. (5 points)**

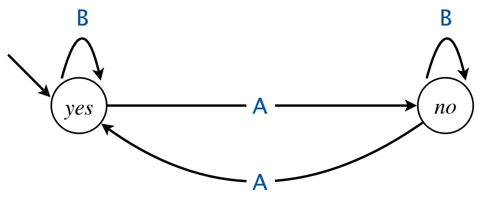
Consider the following regular expressions and DFAs over the binary alphabet $\{A, B\}$.

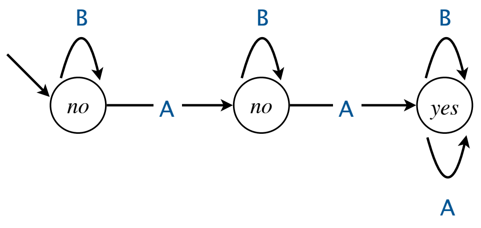
For each regular expression or DFA on the left, write the letter of the best-matching language (set of strings) on the right. You may use each letter once, more than once, or not at all.

☐ `(A | B)* A (A | B)* A (A | B)*`

☐ `(A A)*`

☐ `(A | B)* A A (A | B)*`

☐ 

☐ 

A. number of As is *exactly* 2; no Bs

B. number of As is *exactly* 2; any number of Bs

C. number of As is *even*; no Bs

D. number of As is *even*; any number of Bs

E. number of As is *at least* 2; no Bs

F. number of As is *at least* 2; any number of Bs

G. contains *two consecutive* As; possibly other As and Bs

9. **Intractability and computability. (6 points)**

Suppose that ProblemX is a search problem (i.e., in **NP**).

Mark each statement as either true, false, or unknown by filling in the appropriate bubble.

true	*false*	*unknown*	
●	○	○	Factor poly-time reduces to Sat.
○	○	○	Factor can be solved in exponential time.
○	○	○	Sat can be solved in poly-time but Tsp cannot.
○	○	○	Every problem in **NP** poly-time reduces to Sat.
○	○	○	The halting problem can be solved in exponential time with a Java program on a Macbook Pro running OS X.
○	○	○	If a search problem can be solved in poly-time on a TOY machine, then it can be solved in poly-time on a Turing machine.
○	○	○	$\mathbf{P} \neq \mathbf{NP}$.

10. **Boolean logic and circuits. (6 points)**

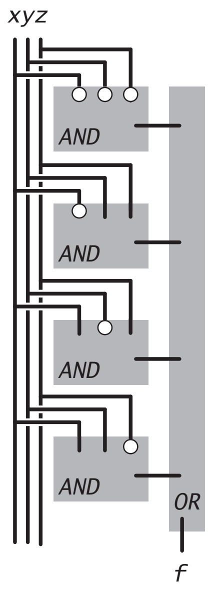
The 3-bit *minority* function $f(x, y, z)$ is a boolean function that is 1 if *at most one* of its three inputs is 1, and 0 otherwise. Which of the following represent the minority function?

Mark all that apply.

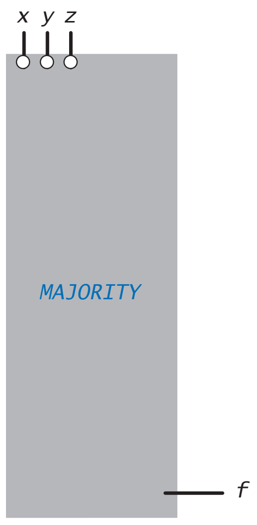
yes ◯ *no* ◯

x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

yes ◯ *no* ◯



yes ◯ *no* ◯



yes ◯ *no* ◯

$$f = x'y'z' + x'y'z + x'yz' + xy'z'$$

yes ◯ *no* ◯

```
public static boolean f(boolean x, boolean y, boolean z) {
   return ! ((x && y) ^ (x && z) ^ (y && z));
}
```

yes ◯ *no* ◯

```
public static boolean f(boolean x, boolean y, boolean z) {
    int count = 0;
    if (x) count++;
    if (y) count++;
    if (z) count++;
    return count <= 1;
}
```