COS 126	Princeton University	Spring 2024

Written Exam 2

This exam has 1010_2 questions worth a total of 64_{16} points. You have 80_{10} minutes.

Instructions. This exam is preprocessed by computer. Write neatly, legibly, and darkly. Put all answers (and nothing else) inside the designated answer 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 are posted is a violation of the Honor Code.

Please complete the following information now.

Name:

NetID:

Exam room: ○ McCosh 10 ○ McCosh 50 ○ McCosh 66 ○ Other

Precept:

P01	P02	P03	P03A	P03B	P04	P04A	P05	P05A	P06
○	○	○	○	○	○	○	○	○	○
P10	P10A	P10B	P11	P12	P13	P14	P14A	P15	P15A
○	○	○	○	○	○	○	○	○	○

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

Signature

1. **Java keywords. (10 points)**

   For each Java keyword on the left, write the letter of the best-matching description from the right. Use each letter at most once.

	Keyword		Description
☐	`public`	**A.**	Defines a data type.
☐	`static`	**B.**	Creates an object.
☐	`void`	**C.**	Specifies that a method has no return type.
☐	`throw`	**D.**	Indicates that a method can be called from another file.
☐	`final`	**E.**	Provides a reference to the object whose instance method (or constructor) is being called.
☐	`new`	**F.**	Provides a reference that does not refer to any object.
☐	`Item`	**G.**	Tells Java where to start the execution of a Java program.
☐	`this`	**H.**	Facilitates access to a library, such as `java.awt.Color`.
☐	`class`	**I.**	Prevents a variable from being modified after initialization.
☐	`import`	**J.**	Identifies a method or variable that belongs to the entire class (and not to an individual object).
		K.	Creates a custom error that typically leads to a run-time exception.
		L.	*Not a Java keyword.*

2. **Using data types. (9 points)**

Assume that the variables `s` and `t` are initialized as follows:

```
String s = "ABABABABAB";
String t = "BABABABABA";
```

For each Java expression on the left, write the letter of the best-matching value (or description) from the right. Use each letter once, more than once, or not at all.

	Expression
☐	`s.length() == t.length()`
☐	`s.substring(1, 2)`
☐	`s.substring(1, 2) = t.substring(2, 3)`
☐	`s.substring(1, 2) == t.substring(2, 3)`
☐	`s.substring(1, 2).substring(2, 3)`
☐	`(s + "A").equals("A" + t)`
☐	`s.charAt(1)`
☐	`s.charAt(4).charAt(1)`
☐	`t.charAt(0) == s.charAt(s.length())`

A. `true`

B. `false`

C. `'A'`

D. `'B'`

E. `"A"`

F. `"B"`

G. `"AB"`

H. `"BA"`

I. `""`

J. `null`

K. *run-time exception*

L. *compile-time error*

3. **Creating and designing data types. (12 points)**

   Recall the `Tracker` data type from Programming Assignment 5. Initially, *lost mode* is disabled. Lost mode can be enabled by calling `enableLostMode()`.

   Consider the following partial implementation of a data type that represents a group of trackers:

```
public class GroupOfTrackers {
    private Tracker[] trackers;
    private int n;

    public GroupOfTrackers(Tracker[] trackers) {
        this.trackers = trackers;
        this.n = trackers.length;
    }

    public int numberInLostMode() {
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (trackers[i].isInLostMode())
                count++;
        }
        return count;
    }

    ...

    public static void main(String[] args) {
        Tracker a = new Tracker("watch", 40.34, -74.66);
        Tracker b = new Tracker("drone", 39.91, -77.02);
        Tracker c = new Tracker("phone", 59.22,  24.48);
        Tracker d = new Tracker("purse", 36.30, -60.00);
        Tracker[] trackers = { a, b, c, d };
        GroupOfTrackers x = new GroupOfTrackers(trackers);

        // MISSING CODE FRAGMENTS ON FACING PAGE

        StdOut.println(x.numberInLostMode());

    }
}
```

Suppose that you run the `main()` on the facing page with each of the code fragments below inserted into the designated place. What will be printed to standard output?

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

☐
```
a.enableLostMode();
b.enableLostMode();
```

☐
```
a.enableLostMode();
b = a;
b.enableLostMode();
a.disableLostMode();
```

☐
```
b.enableLostMode();
c.enableLostMode();
trackers[3] = a;
trackers[2] = d;
a.enableLostMode();
c.disableLostMode();
```

☐
```
trackers = new Tracker[6];
trackers[4] = a;
trackers[5] = b;
a.enableLostMode();
d.enableLostMode();
```

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

G. 6

H. *run-time exception*

4. **Machine learning. (9 points)**

Consider the following API for `BinaryClassifier`, which predicts whether a new patient will stay in a hospital for longer than 15 days based on the patient's vital signs at the time of admission.

```
public class BinaryClassifier {

    // creates a new BinaryClassifier
    public BinaryClassifier()

    // trains the classifier on the given file
    public void train(String filename)

    // returns the error rate on the given file
    // (0.0 for no errors, 1.0 for all errors)
    public double test(String filename)

}
```

You are provided with two files, `testing.txt` and `training.txt`. Each line in the file contains the vital signs of a patient at the time of admission, followed by +1 if the patient stayed longer than 15 days, and −1 otherwise. The files contain disjoint groups of patients, and all patients in `testing.txt` were admitted after the patients in `training.txt`.

For each client below, determine whether the binary classifier is *learning* to make correct predictions for *new* patients.

For each client, fill in the best-matching bubble.

(a)

client code	*return value*
`BinaryClassifier x = new BinaryClassifier();`	–
`x.train("training.txt");`	–
`double error = x.test("testing.txt");`	`0.1`

○ learning ○ *not* learning ○ cannot be determined

(b)

client code	*return value*
`BinaryClassifier x = new BinaryClassifier();`	–
`double error1 = x.test("training.txt");`	`0.3`
`x.train("training.txt");`	–
`double error2 = x.test("training.txt");`	`0.0`

○ learning ○ *not* learning ○ cannot be determined

(c)

client code	*return value*
`BinaryClassifier x = new BinaryClassifier();`	–
`double error1 = x.test("testing.txt");`	`0.3`
`x.train("training.txt");`	–
`double error2 = x.test("testing.txt");`	`0.1`

○ learning ○ *not* learning ○ cannot be determined

```
                     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


 16 16-bit registers:       R[0] to R[F]
256 16-bit memory locations: M[00] to M[FF]
  1  8-bit program counter:  PC

R[0] always reads as 0000.
Loads from M[FF] come from stdin.
Stores to  M[FF] go to stdout.
```

5. **TOY programming. (8 points)**

(a) Which value is stored in `R[C]` upon termination of the following TOY program?

Fill in the best-matching bubble.

```
10: 7A11
11: 8B12
12: AC0A
13: 0000
```

○ 0000 ○ 0011 ○ 0012 ○ 000A ○ 7A11 ○ 8B12 ○ AC0A

(b) What does the following TOY program print to standard output?

Fill in the best-matching bubble.

```
10: 7ACA   R[A] = 00CA
11: 7B14   R[B] = 0014
12: EB15   PC <- R[B]
13: 9AFF   write R[A]
14: 9BFF   write R[B]
15: FB12   R[B] <- PC; PC <- 12
16: 9AFF   write R[A]
17: 0000   halt
```

○	○	○	○	○	○	○
nothing	0014	0014 00CA	0014 0014 0014 0014 ⋮	0014 0016 0016 0016 ⋮	00CA 0014	00CA 00CA 00CA 00CA ⋮

6. **TOY machine. (12 points)**

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

☐	Number of 1s in the binary representation of the decimal integer 226.
☐	Number of 1s in the binary representation of −1. *Assume 16-bit two's complement integer.*
☐	Total number of *bits* of main memory in TOY (including memory location `FF`).
☐	Number of *bytes* that the TOY program counter stores.
☐	Number of distinct *negative* integers representable in Java's `int` data type.
☐	Value of the Java expression: `(-8192 ^ -1) + 1`

A. 2^0 (1)

B. 2^1 (2)

C. 2^2 (4)

D. 2^3 (8)

E. 2^4 (16)

F. 2^5 (32)

G. 2^6 (64)

H. 2^7 (128)

I. 2^8 (256)

J. 2^9 (512)

K. 2^{10} (1,024)

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)

R. 2^{20} (1,048,576)

S. 2^{31} (2,147,483,648)

T. 2^{32} (4,294,967,296)

7. Algorithms. (10 points)

Implement a version of *binary search* that determines whether a key appears in an array whose elements are sorted in *descending order*.

(a) Complete the implementation of `containsKey()` by, for each oval numbered 1–5, choosing one of the expressions from the right. No other code is allowed.

Your answer should be a sequence of uppercase letters (corresponding to the labeled ovals). Use each letter at most once.

```
public static boolean containsKey(int[] a, int key) {
    int lo = [ 1 ] ;
    int hi = [ 2 ] ;
    // if key is in a[], it is in the subarray a[lo..hi]
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid])  [ 3 ] ;
        else if (key > a[mid])  [ 4 ] ;
        else return [ 5 ] ;
    }
    return false;
}
```

- **A.** `false`
- **B.** `true`
- **C.** `-1`
- **D.** `0`
- **E.** `1`
- **F.** `a.length - 1`
- **G.** `a.length`
- **H.** `lo = mid`
- **I.** `lo = mid - 1`
- **J.** `lo = mid + 1`
- **K.** `hi = mid`
- **L.** `hi = mid - 1`
- **M.** `hi = mid + 1`

1	2	3	4	5

(b) Suppose that the length of `a[]` is 1 million and `key` does *not* appear in `a[]`. Assuming the binary search is implemented correctly, approximately how many times will the body of the `while` loop execute?

Fill in the best-matching bubble.

○ 1 ○ 10 ○ 20 ○ 30 ○ 40 ○ 1,000 ○ 1,000,000

8. **Data structures. (10 points)**

Suppose that the following code fragment is used to initialize `s`, `queue`, and `st`:

```
String s = "BAABCA";
Queue<Integer> queue = new Queue<Integer>();
ST<Character, Stack<Integer>> st = new ST<Character, Stack<Integer>>();

for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    if (!st.contains(c)) {
        st.put(c, new Stack<Integer>());
        queue.enqueue(i);
    }
    st.get(c).push(i);
}
```

For each code fragment on the left, write the letter of the best-matching output from the right. Use each letter at most once.

```
StdOut.println(st.size());
```

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

```
for (char c : st.keys()) {
    StdOut.print(st.get(c).size() + " ");
}
```

```
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    StdOut.print(st.get(c).pop() + " ");
}
```

A. 3

B. 5

C. 0 1 4

D. 1 2 3

E. 2 3 1

F. 3 2 1

G. 4 1 0

H. 0 1 2 3 4 5

I. 1 1 1 1 1 1

J. 3 5 2 0 4 1

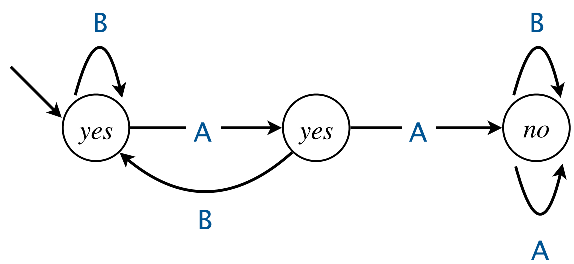
K. 3 5 5 3 4 5

L. 5 4 3 2 1 0

9. **Theory of computing. (10 points)**

(a) Describe the set of strings that the following DFA matches.

Fill in the bubble corresponding to the best-matching description.



◯ starts with AA

◯ ends with AA

◯ contains AA

◯ does *not* start with AA

◯ does *not* end with AA

◯ does *not* contain AA

(b) Identify each statement below as *known to be true*, *known to be false*, or *unknown*.

true	*false*	*unknown*	
◯	◯	◯	If you can design a Turing machine to solve some computational problem X, then you can write a Java program to solve X.
◯	◯	◯	If you can write a Java program to solve some computational problem X, then you can design a Turing machine to solve X.
◯	◯	◯	The Java compiler can identify all Java statements in a program that are *unreachable* (i.e., statements that can never be executed because there is no way for the program flow to reach it).
◯	◯	◯	The Java compiler can determine whether a given `.java` file is a *syntactically valid Java program*.
◯	◯	◯	It is possible to build a computational device based on solar eclipses that can solve the *halting problem*.
◯	◯	◯	There exists a computational problem that can be solved on a DFA but not on a Turing machine.
◯	◯	◯	A *universal Turing machine* is a Turing machine that can solve the halting problem.

10. **Digital circuits. (10 points)**

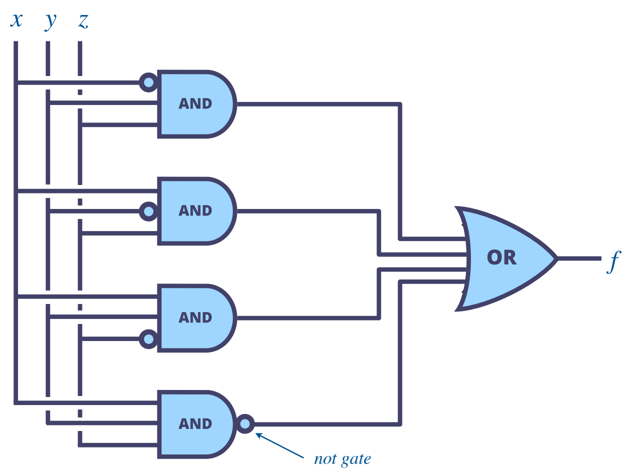
The 3-bit *even-parity* function $f(x, y, z)$ is a boolean function of 3 variables that is 1 if an even number of its arguments are 1, and 0 otherwise. Which of the following represent the 3-bit even-parity function?

Fill in the corresponding bubbles.

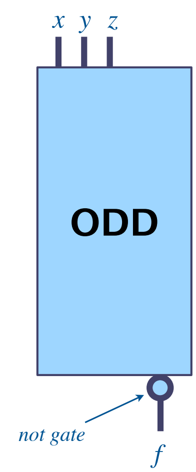
yes ◯ *no* ◯

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

yes ◯ *no* ◯



yes ◯ *no* ◯



yes ◯ *no* ◯ $f = x'y'z' + x'yz + xyz'$

yes ◯ *no* ◯

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