CSCI-UA.0101-007: Final Exam

Duration: 90 minutes

Instructor: Michael Tao
New York University

December 20, 2023

Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, use the extra pages at the end of the exam.

Name: _____

NetID: _____

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED.

Instructions:

- **Write your full name and your NetID on the front of this exam.**
- Make sure that your exam is not missing any sheets. There should be (17) double sided pages in the exam.
- Write your answers in the space provided below each problem. If you make a mess, clearly indicate your final answer (or state that it is on the scrap paper).
- If you have any questions during the exam, raise your hand and we will get to you.
- At the end of the exam, there is one blank page. Use this as your scrap paper. If you need additional scrap paper, raise your hand and we will get it for you.
- This exam is closed books, closed notes, closed computers. You are allowed to use a calculator (*but not your cellphone*) to answer the numerical questions (if any).

Good luck!

Problem 1: Just Functions

In this section just implement functions - do not worry about `static` or classes.

(1) One of the most common "nonlinearities" used in neural networks is the Rectified Linear Unit, which is defined as $ReLU(x) = x$ if x is positive and 0 otherwise. Implement the `ReLU` function that uses a floating point type (pick whichever you want).

(2) Say we have the class

```
public class FileLoader {

    // loads a .doc file
    public static void loadDoc(String path) {...}
    // loads a .txt file
    public static void loadTxt(String path) {...}
    // loads a .png file
    public static void loadPng(String path) {...}
}
```

Implement a function called `load` that takes in a `String` representing a path, determines its extension (whether it ends with `".doc"`, `".txt"`, or `".png"`), and finally invokes the appropriate loader function using a `switch/case` statement.

(3) Spin locks are a standard tool for controlling access to resources in parallel computing. They control access to a resource that we want to obtain so that users don't accidentally access resources that are unavailable, such as when another user is utilizing them or if the resource is not available yet (like a file being downloaded).

To access a resource guarded by a spin lock, the user utilizes `lock` function which will sometimes fail. When `lock` fails it returns `null` instead of a `Resource` object. If a user decides they really want to acquire a resource they resource they will repeatedly try to call the `lock` function until it doesn't return `null`.

You are tasked with writing an `acquire` that takes in a `SpinLock` object and repeatedly tries to acquire the resource by calling the `lock` member function. Once the resource is acquired it returns the `Resource` object.

```
public class SpinLock {
    // returns null until the resource is available.
    public Resource try_lock();
}
```

(4) Implement a function called `outerProduct` that takes in two arguments `a`,`b` of type `double[]` and returns a multiplication table based off of those numbers. In particular, return a two dimensional array such that each entry $A[i][j]$ is the product of inputs $a[i]$ and $b[j]$.

(5) **Background:** Run-length encoding is a fundamental, yet powerful, technique for reducing the amount of space used to store data, especially when data being stored contains long consecutive sequences of a single value. The run-length encoding of an array of `boolean` values is an alternating sequence of integral numbers that indicate how many 1s or 0s occur in a row. With an assumption that every sequence starts with a 0, the decode function for a run-length encoded string should produce output sequences like:

$$decode(\{5\}) = \{0, 0, 0, 0, 0\} \tag{1}$$

$$decode(\{2, 3\}) = \{0, 0, 1, 1, 1\} \tag{2}$$

$$decode(\{1, 2, 9, 3\}) = \{0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1\} \tag{3}$$

$$decode(\{0, 10\}) = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\} \tag{4}$$

Even though we will assume that every sequence starts out with 0s, we can simply say that there are 0 0s to make a sequence that starts with a 1 (as is seen in the last example above).

Implement a `decode` function that decodes a run-length encoded sequence. The input should be an array of integral numbers and the output should be an array of boolean values. You must evaluate the total length of the final decoded data as part of your algorithm.

Problem 2: Ballots

In this problem we will implement a class that accumulates a binary ballot scheme on some motion, including abstentions. That is, people submit a single string that says either `"yes"`, `"no"`, or `"abstain"` to declare their opinion on whether the motion should be passed. Invalid submissions can also be tossed out.

We will make it so that we cannot extract results unless the ballot is locked. Otherwise it's possible to determine how voters voted during the voting process.

(1) Declare a class called `Ballot`

```
{
```

(2) Declare members that hold the number of `"yes"`,`"no"`, and `"abstain"` received. Also, store the number of votes that were tossed and a variable to indicate if the ballot is locked.

(3) Implement getters for each of the members that you defined, but we will throw a checked exception called `InvalidAccessException` for the number of `"yes"` and `"no"` responses. Feel free to simply write `IAE` as shorthand for this exception's name (as it is a long word to write).

(4) Implement a function called `makeLocked`, which locks the ballot.

(5) Declare a constructor that assigns appropriate default values to each member.

(6) Implement a member function called `process` that processes a vote by taking in a `String` and updates the class members according to the argument with a `switch/case` statement. This function will be strict - the user needs to pass in exactly what the strings listed above (lower case, no extra spaces, etc) or the vote will be tossed. A vote will also be tossed if the ballot is locked.

(7) Implement another member function, also called `process` that takes in an array of votes, each represented by a `String`, and processes each of those votes.

(8) Implement a constructor that takes in an array of votes and properly populates the members of the class.

(9) Implement a member function `result` that reports the result of the ballot.

1. If the vote is not locked then it will return `null`.
2. If there were more `"yes"` than `"no"` it will return `"yes"`
3. If there are more `"no"` than `"yes"` then it will `"no"`.
4. If there are an equal number of `"no"` and `"yes"` then it will `"tie"`.

(10) Implement a member function `voteCount` that reports the total number of votes that were received.

(11) Implement a member function `validVoteCount` that reports the number of yes or no votes that were received.

```
}
```

Problem 3: Middleware

When working on our own software it is quite common to find other software libraries that already implement functionality that we need. We often cannot (and should not) copy-paste this code, but rather need to call functions that they provide or satisfy interfaces that they provide. One such circumstance is where you mostly need to implement some interface that simply invokes some other interface to connect two different libraries together, which is part of writing *middleware*.

In this problem we have one library that uses datasets to potentially to compute statistics and another library for storing data, a database. You will write some middleware that creates a sort of dataset that uses a database to read and write data.

```
public interface DatabaseInterface {
    double get(int userIndex, int columnIndex);
    void write(int userIndex, int columnIndex, double value);

    int getUserIndex(String userName);
    int getColumnIndex(String columnName);

    String[] getUserNames();
    String[] getColumnNames();
}
```

```
public interface DatasetInterface {
    double get(String userName, String columnName);
    void write(String userName, String columnName, double value);

    double[] getRow(String userName, String[] columnNames);

    // returns a table of values for every userName/columnName provided
    double[][] getTable(String[] userNames, String[] columnNames);

    // returns a table of values for every userName and columnName.
    // The data is arranged as if the userNames and columnNames were in
    // alphabetical order
    double[][] getTable();
}
```

(1) Declare a class called `DatabaseDataset` that takes on the `DatasetInterface`.

(2) Declare any members that might be appropriate to make sure we can access a database.

(3) Declare a constructor that takes in a `DatabaseInterface` object and initializes members.

(4) Implement `get`

(5) Implement `write`

(6) Implement `getRow`

(7) Implement `getTable` with arguments.

(8) Implement `getTable` without arguments. Note that the `getUserNames` and `getColumnNames` do not guarantee that they return sorted arrays, but this function is specified to want sorted names.

Problem 4: Baguettes

`Background:` Optimal transport formalizes the challenge of finding the most efficient way to distribute resources. It generalizes the concept of distance between two individual entites (like a person and a bakery) to describe *average* distances between two populations or distributions. For this problem we are interested in evaluating the average distance a person must walk to reach their bakery, which is sometimes called the ℓ_1 Wasserstein distance.

Say we are given a set of bakeries in a town that only sell baguettes and for each bakery we are given a list of dedicated customers who only buy one baguette a day from one bakery. The Wasserstein distance here is the average distance each person in the town has to walk to pick up a baguette from their bakery. If these bakeries and customers all live in a grid-based town (like Manhattan) then the distance between any customer and a bakery is best implemented by the ℓ_1 distance as well. For this problem we will specify the coordinates of bakeries and customers with this `Locatable` interface:

```
public interface Locatable {
    double getX();
    double getY();
}
```

Here's an interface to satisfy for the bakery:

```
public interface BakeryInterface extends Locatable {
    Customer[] getCustomers();
    void addCustomer(Customer c);
}
```

(1) Declare a `Customer` class that uses the `Locatable` interface.

(2) Declare members to store the Customer's location (their x and y coordinates).

(3) Implement a constructor that takes in the customer's location.

(4) Finish implementing the interface.

(5) Declare a sort of bakery class called `WassersteinBakery` that does not have every function implemented and therefore cannot be instantiated. Even though you will be implementing some functions, you will not be implementing any functions defined in the `BakeryInterface`.

(6) Implement a function called *l*1*Distance* that takes in a `Locatable` and evalautes the ℓ_1 distance between the current bakery and its argument. Recall that the ℓ_1 distance is defined by

$$d_{\ell_1}(a, b) = |a_x - b_x| + |a_y - b_y|. \tag{5}$$

Feel free to use `Math.abs`.

(7) Implement a member function called `wassersteinDistance` that does not require constructing an instance of a `WassersteinBakery` to invoke. The function takes in an array of `WassersteinBakery` objects and computes the average distance traversed by customers over all of the customers of each bakery.

Scrap paper **Name**: _____

Scrap paper **Name**: _____

Scrap paper **Name**: _____

```
// Some useful java functions

int[][] x = new int[4][];
x[3] = new int[]{1,2,3};

int[][] y = new int[4][3];

public class Arrays {
    // returns a new array with the data sorted
    int[] sort(int[] values);
    // not quite in the true API, but lets assume this exists
    String[] sort(String[] values);
}



class String {
    int length();
    String[] split(String regex);// just need to pass "." for this exam
    boolean equals(String other);

}
```