C212 Final Exam (150 points)
December 19, 2025

PLEASE READ ALL DIRECTIONS BEFORE STARTING YOUR EXAM. DO NOT OPEN UNTIL YOU ARE TOLD TO DO SO.

This is a closed-note exam except for your approved aid sheet. No electronic devices or communication with anyone other than proctors and the professor is allowed. *Cheating or talking with others will result in an automatic F in the course.*

Unless a question states otherwise, you may use any Java class without writing its corresponding `import`. You do not need to write the full design recipe (signature, documentation, tests), though you may do so if it helps.

If you believe you have found an error on the exam, raise your hand and notify a proctor.

When you are finished, check over your work. If more than 10 minutes remain, submit your exam and aid sheet to a proctor, show your ID, then quietly exit. No one may leave during the final 10 minutes.

Do not tear out, detach, or remove any pages from this exam. If you use the scratch sheet of paper, please label the question on the sheet and put a note on the question.
Do not waste time erasing; simply cross out your work and we will ignore it.

Illegible hand-writing will receive a 0.

The exam contains 150 points, plus up to 20 points of extra credit (170 possible).

You have **120 minutes** to complete the exam.

Good luck!

Question	Points	Score
1	70	
2	30	
3	50	
4	0	
Total:	150	

Name: ______________________________

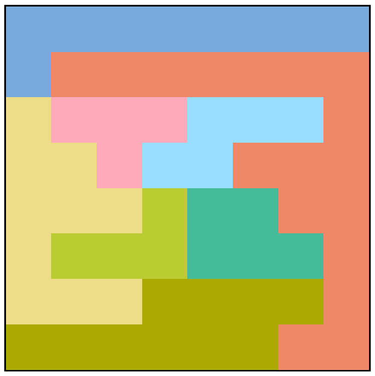
IU Email: ______________________________

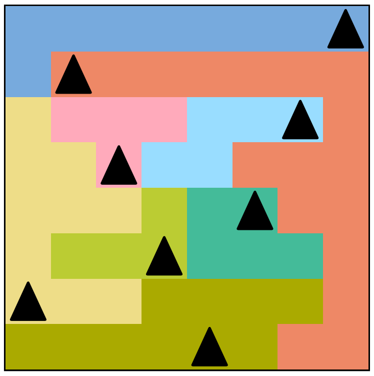
Part I

Recommended Time: 45 minutes

1 Problem

1. (70 points) LinkedIn has a game called "queens," where the objective is to place a single queen in each "sub-board," denoted by a different color, such that no queen attacks another. A queen attacks another if it (1) is in the same row, (2) is in the same column, or (3) is diagonal but touching. In the left figure below, we show an unsolved "queens" game. In the right figure below, we show the satisfying queens placement.





In this question, you will implement an algorithm that solves this problem.

Assume the existence of the following two class definitions. These classes contain omitted implementations of `equals`, `hashCode`, and `toString`.

```java
/**
 * A Position is a (x, y) coordinate pair.
 */
final class Position {

  private final int X;
  private final int Y;

  Position(int x, int y) {
    this.X = x;
    this.Y = y;
  }

  int getX() { return this.X; }
  int getY() { return this.Y; }
}
```

```java
/**
 * A Board is a collection of lists of positions. The separate lists of positions
 * correspond to a sub-board. Namely, Boards contain sub-boards.
 */
final class Board {

  private final List<List<Position>> POSITIONS;

  Board(List<List<Position>> posns) {
    this.POSITIONS = posns;
  }

  List<Position> getSubBoard(int idx) { return this.POSITIONS.get(idx); }

  List<List<Position>> getAllSubBoards() { return this.POSITIONS; }
}
```

(a) *(8 points)* First, design the `boolean hasQueenConflict(Set<Position> P, Position q)` method that returns whether placing a queen at q conflicts with any existing queens in P. Assume that there exists a method called `isDiagonalTo(p, q)`, which returns whether two queens are diagonally adjacent to one another. **Your implementation must use `isDiagonalTo` to receive full credit.**

```
private static boolean hasQueenConflict(Set<Position> P, Position q) {

}
```

(b) *(12 points)* Second, design the `boolean allSubBoardsPopulated(Board B, Set<Position> queens)` method, which returns whether all of the sub-boards in a given `Board` have exactly one queen placed in them. You may assume that a sub-board has at most one queen.

```
private static boolean allSubBoardsPopulated(Board B, Set<Position> queens) {

}
```

(c) *(30 points)* Third, finish designing the following methods. The `solveHelper` method processes the sub-boards of B one at a time.

Its base case is simple: if we have populated every sub-board, return V. Otherwise, consider the sub-board at index idx (starting from 0). For each candidate position p in the sub-board, attempt to place a queen at p. If that placement introduces no conflicts with the existing queens in V, recursively solve the next sub-board (i.e., index $idx+1$) with the updated set of queen positions V with p added (i.e., $V \cup \{p\}$). If that recursive call returns a non-`null` result, then we have successfully placed queens in all sub-boards and can return that result. Oppositely, if that recursive call returns `null`, undo the placement and try the next candidate position in the current sub-board. Finally, if no candidate positions in the current sub-board lead to a solution, return `null`.

Hint: do not get intimidated by this method. READ THE DIRECTIONS!

```
static Set<Position> solve(Board B) {
  return solveHelper(________________________________________________);
}

private static Set<Position> solveHelper(Board currBoard,
                                         Set<Position> queens,
                                         int idx) {
  // Once we all of the boards are populated, we return the queens set.
  if (____________________________________) {
    return queens;
  } else {
    // For every position in the current board...
    for (_____________ p : currBoard.getSubBoard(_______)) {
      if (!hasQueenConflict(________________________________________)) {

        // Add the current position to the queens set.
        queens.add(_____);
        Set<Position> result = solveHelper(________________________________________);

        // If we found a non-null result, that's the answer, so return it.
        // Otherwise, remove the position we tried from the queens set.
        if (result != null) {
          return ____________;
        } else {
          __________.remove(__________);
        }
      }
    }
    return null;
  }
}
```

(d) *(5 points)* What is the asymptotic runtime of `hasQueenConflict` in the worst case? Assume n is the number of queens in the given set, and further assume that `isDiagonalTo` runs in constant time.

(e) *(5 points)* What is the asymptotic runtime of `allSubBoardsPopulated` in the best case? Assume that set queries run in constant time, and that n represents the number of sub-boards in the board and m represents the number of queens in the set `queens`.

(f) *(5 points)* Big-Omega represents the ______________ bound on the growth of a function.

(g) *(5 points)* Big-Oh represents the ______________ bound on the growth of a function.

Part II

Recommended Time: 75 minutes

2 Problems

2. (30 points) This question has three parts, each of which are weighed equally.

   You're on the planet Lauris, and you're armed with a high-powered telescope. Your telescope reads distances in light years from Lauris to different stars. You want to figure out the closest/minimum distance that is both "relevant" and "enlightening."

   - A distance is "irrelevant" if it is greater than 1,000 light years away.
   - A distance is "enlightening" if it is divisible by 7.

   For example, suppose $D = [974, 1101, 1000, 34, 28, 14, 83]$. After skipping over the irrelevant distances and keeping the enlightening distances, we have $[28, 14]$. The minimum is 14 light years.

   For all problems, assume there is at least one relevant and enlightening distance in D.

   For all problems, assume all distances in D are positive integers.

   For all problems, you are not allowed to "pre-process the data." That is, you cannot filter out the "bad distances," and then recursively/iteratively find the minimum distance. You must use exactly one traversal over the list or credit will not be given.

   (a) Design the *standard recursive* `static int minDistance(List<Integer> D)` method that returns the minimum distance using the above criteria. You will likely need to design a helper method. **Your method must be standard recursive or it will receive 0 points.**

(b) Design the *tail recursive* `static int minDistanceTR(List<Integer> D)` method that uses tail recursion to solve the problem. You will need to design a helper method. Remember to include the relevant access modifiers! **Your helper method must be tail recursive or you will receive 0 points.**

(c) Finally, design the *iterative* `static int minDistanceLoop(List<Integer> D)` method that uses a loop to solve the problem. **Your method must be iterative with zero recursive calls or you will receive 0 points.**

Before proceeding...

For the last question, did you use the translation pipeline to solve the problem? It was not required; I am just curious!

__________ Yes

__________ No

3. (50 points) In this question you will design a series of classes to represent an online learning platform.

   (a) *(6 points)* First, design the **complete** `Gradable` interface. It should contain two methods: `double grade()` and `boolean isComplete()`. The former represents the grade from 0-100, and the latter returns whether the work is fully submitted.

(b) *(16 points)* Design the **complete** abstract `Submission` class, which implements `Gradable`. It stores the name of the submission and the name of the student who submitted it. The constructor should receive these as arguments and assign them to instance variables. Override the methods from the interface by declaring them as abstract.

Override `equals` as follows: two `Submissions` are equal if they have the same name, have the same student submitting them, have the same grade, and have the same completion status.

Override `hashCode` as follows: compute the hash code of the submission name and the student name using `Objects.hash`.

(c) *(8 points)* Design the abstract `AutoGraded` class, which extends `Submission`. This class represents an autograded submission. It stores a maximum number of points as an instance variable. Its constructor receives the submission name, the name of the student, and the maximum number of points. Call the superclass constructor appropriately. Do not override any other methods.

Create an accessor for the maximum number of points instance variable.

```
_______________ class AutoGraded _________________ Submission {

  ___________ final ____________ MAX_POINTS;

  Autograded(______________________________________________________) {
    ___________________________________________;
    this.MAX_POINTS = _____________________;
  }

  ____________ getMaxPoints() {
    __________ this.MAX_POINTS;
  }
}
```

(d) *(10 points)* Design the `CodeSubmission` class, which extends `AutoGraded`. It stores a value between 0 and 1 (inclusive on both ends) representing the total code coverage ratio. It also stores a boolean representing whether it passes all of the test cases. The constructor receives the submission name, the name of the student, the maximum number of points, the code coverage ratio value, and whether the submission passed all of the tests. Call the superclass constructor appropriately.

Override `grade` as follows: if the code passed all of the tests, return the maximum score. Otherwise, return the maximum score multiplied by the code coverage ratio.

Override `isComplete` by returning true if they pass all of the tests or the code coverage ratio is at least 0.80. Return false otherwise.

```
class CodeSubmission _________________ _________________ {

  ___________ final ____________ HAS_PASSED_TESTS;
  private final _________ CODE_COVERAGE;

  CodeSubmission(_____________________________________________________________) {
    _____________________________________________________________;
    this.HAS_PASSED_TESTS = ____________________;
    this.CODE_COVERAGE = ____________________;
  }

  @Override
  public double grade() {





  }

  @Override
  public boolean isComplete() {





  }
}
```

(e) *(10 points)* Design the `MultiPartSubmission` class, which extends `Submission`. This submission represents a multi-part assignment, so it will store a `List<Submission>`, designating the parts of the submission. The constructor should receive the submission name, the student's name, and a `List<Submission>`. Call the superclass constructor appropriately. Assign the list over to the instance variable.

Override `grade` as follows: if `PARTS` is empty, throw an `IllegalStateException`. Otherwise, return the average of all of the submissions' grades after calling `grade` on each one.

Override `isComplete` as follows: return true if all of the submissions in `PARTS` are complete and false otherwise.

```
class MultiPartSubmission ________________ _________________ {

  ___________ final ___________________________________ PARTS;

  MultiPartSubmission(_______________________________________________________________) {
    _______________________________________________________________;
    this.PARTS = ____________________;
  }

  @Override
  public double grade() {









  }

  @Override
  public boolean isComplete() {









  }
}
```

4. (0 points) This question has no required parts. Answering any of the following questions awards extra credit. You should use only the space provided to write your answer; anything more embellishes upon what we're looking for.

   (a) *(4 extra credit points)* We know that binary search runs in $\Theta(\lg n)$ in the worst case. Assume Java programs stack overflow after $8,000$ recursive calls. What is the minimum number of elements that an array needs to be for a correct binary search implementation to stack overflow? **This answer should be numeric; not asymptotic. Do not simplify your answer.**

   (b) *(3 points)* Provide a *programming* example of concurrency and a *programming* example of parallelism.

   (c) *(3 points)* What is the most important thing that you learned this semester in C212?

More extra credit is on the next page.

(d) *(10 points)* For each of the following problems, describe the worst-case runtime of the algorithm. Note that these are all-or-nothing points, so getting the bound *and* the class (i.e., Big-Oh, Big-Omega, Theta) correct are both required! Specify the tightest bound where possible.

(i)
```java
static boolean isPalindrome(String s) {
  for (int i = 0; i < s.length() / 2; i++) {
    if (s.charAt(i) != s.charAt(s.length() - i - 1)) {
      return false;
    }
  }
  return true;
}
```

(ii)
```java
static <T> Optional<Integer> linearSearch(T[] A, T t) {
  for (int i = 0; i < A.length; i++) {
    if (A[i].equals(t)) { // Assume "equals" runs in constant time.
      return Optional.of(i);
    }
  }
  return Optional.empty();
}
```

(iii)
```java
static int tribonacci(int n) {
  if (n <= 1) {
    return n;
  } else if (n == 2) {
    return 1;
  } else {
    return tribonacci(n - 1) + tribonacci(n - 2) + tribonacci(n - 3);
  }
}
```

(iv) `ArrayList` "`remove`."

(v) `LinkedList` "`insert`."

Scratch work