C212 Final Exam (150 points)
May 7, 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 aside from your one page of notes, double-sided. You may not use any electronic devices to complete this exam, nor can you communicate with anyone besides the proctors and professor. *If you are caught cheating, you will receive an F in the course.*

For any question, unless specified otherwise, you may use any class without a corresponding `import`. E.g., if you want to use `HashMap`, you do not need to also import `java.util.HashMap`.

Unless otherwise stated, you do not need to spell out the "full design recipe", i.e., write the signature, documentation comments, and tests. Of course, doing so may aid you in your solution.

If you find a mistake, please raise your hand and let one of the proctors know; we will determine whether or not this is the case.

The exam has 150 total points, with 20 extra credit points for a total possible score of 170/150.

If you need to use the restroom, raise your hand and let a proctor know. You must turn in your exam, cheat sheet, and phone before leaving. You will receive these back upon your return.

When you are finished, check over your work carefully, turn in your exam and notes sheet if you have one, then quietly exit.

You have 120 minutes to complete the exam.

Good luck!

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

Name: _____

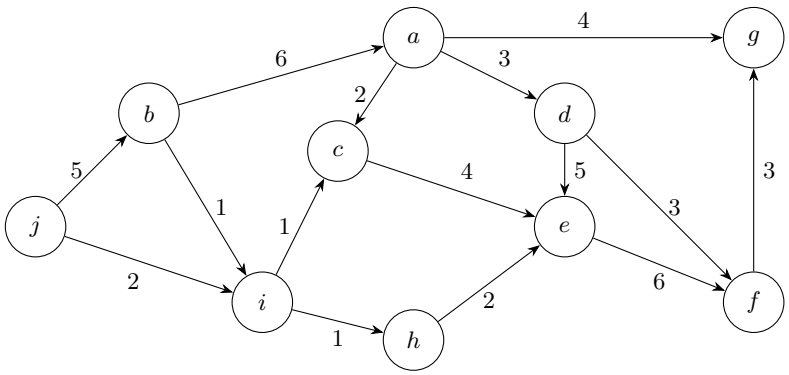
IU Email: _____

Part I

Recommended Time: 60 minutes

1 Problem

1. (70 points) In this question you will design a class that represents a *graph*. A graph is a tuple (V, E) where V represents a set of vertices and a E represents a set of edges that connect vertices with an *edge weight*. Note that this is *not* a Cartesian coordinate graph! An example of a graph G_1 is below.



Where $V = \{a, b, c, d, e, f, g, h, i, j\}$, and $E = \{(a, c, 2), (a, d, 3), (a, g, 4), (b, a, 6), (b, i, 1), (c, e, 4), (d, e, 5), (d, f, 3), (f, g, 3), (h, e, 2), (i, c, 1), (j, b, 5), (j, i, 2)\}$. Note that the elements of E are triples where the first value is the source vertex, the second is the destination vertex, and the third is the weight.

This kind of graph is *directed*, meaning edges have a direction, and *weighted*, meaning there is a "cost" to go from one vertex to another.

There are several ways to implement a graph in code. One way is to simply store a list of edges in the graph. This is a rather inefficient way to do so, but it encodes enough information to represent a connected graph, i.e., where every vertex can be reached from every other vertex.

(a) *(7.5 points)* Design the `Vertex` class. A vertex stores an identifier as a string. The vertex should be immutable, so make it a final variable and design the relevant accessor method.

```
________ Vertex {


  _______________ _______________ _______________ ID;

  Vertex(__________________) {


  }









}
```

(b) *(5 points)* Inside `Vertex`, override the `public boolean equals(Object o)` method for comparing two `Vertex` instances. Two `Vertex` instances are equal if their identifiers are the same.

```
@Override
_______________ _______________ equals(_______________ o) {
  if (_________________________________) {
    return false;
  } else {
    Vertex othVertex = ___________________ o;









  }
}
```

(c) *(7.5 points)* Design the abstract `Edge` class. An `Edge` contains an immutable source `Vertex` and an immutable destination `Vertex` as fields. The constructor should receive these and store them as instance variables. Design the relevant accessor methods.

```
_______________ _______________ Edge {

  _______________ final _______________ SOURCE;
  private _______________ _______________ DESTINATION;

  Edge(_______________ src, _______________ dest) {



  }
}
```

(d) *(5 points)* Inside `Edge`, override the `public boolean equals(Object o)` method for comparing two `Edge` instances. Two `Edge` instances are equal if their source and destination vertices are equal.

```
@Override
_______________ _______________ equals(_______________ o) {
  if (_________________________________) {
    return false;
  } else {
    Edge othEdge = ___________________ o;









  }
}
```

(e) *(10 points)* Design the `WeightedEdge` concrete class, which extends `Edge`. It should store only an integer representing the edge weight. Its constructor should receive the source and destination vertices, as well as the weight of the edge. Write the accessor and mutator methods.

```
_______________ WeightedEdge _______________ ____________ {

  _______________ _______________ weight;

  WeightedEdge(_______________ src, _______________ dest, _______________ weight) {
    __________(______________________);


  }









}
```

(f) *(5 points)* Inside `WeightedEdge`, override the `public boolean equals(Object o)` method for comparing two `WeightedEdge` instances. Two `WeightedEdge` instances are equal if their source, destination, and weights are equal. **You must call the superclass' version of `.equals` to receive credit.**

```
@Override
_______________ _______________ equals(_______________ o) {
  if (_________________________________) {
    return false;
  } else {
    WeightedEdge othWEdge = ___________________ o;









  }
}
```

(g) *(5 points)* Design the `IGraph` interface, which represents a graph. It contains two methods: one for retrieving a `Set<Vertex>` and another for retrieving a `Set<Edge>`.

```
_______________ IGraph {

  ___________________________________ vertices();


  ___________________________________ edges();
}
```

(h) *(15 points)* Design the `WeightedGraph` class. It should store a `List<Edge>` representing the edges of the graph. The constructor should instantiate this list as a new `ArrayList`.

Implement the `vertices()` and `edges()` methods with the appropriate access modifiers and return types. Inside the methods, return the object as a `HashSet`. (You may assume that `Vertex` and `Edge` have the `hashCode` method overridden, even if you did not write it yourself.)

Design the `void addEdge(Vertex s, Vertex d, int w)` method that adds an `Edge` instance with the source vertex s, destination vertex d, and weight w.

Design the `void addEdge(Vertex s, Vertex d)` method that adds an edge from vertex s to vertex d with weight 1 to the graph. **You must call the other version of `addEdge` to receive credit.**

THE SKELETON CODE IS ON THE NEXT PAGE.

```
_____________ WeightedGraph _____________ IGraph {

  ____________ __________________________ edgesOfGraph;

  WeightedGraph() {
    this.edgesOfGraph = _________________________________;
  }

  @Override
  ________________________ ________________________ vertices() {









  }

  @Override
  ________________________ ________________________ edges() {









  }

  void addEdge(_____________________________________________________) {


  }

  void addEdge(_____________________________________________________) {


  }
}
```

(i) *(5 points)* While I was writing this question, for a brief moment, I thought, "I should add a class called `UnweightedEdge`; we can then add unweighted edges to a graph." But then I realized that is unnecessary. Why did I come to that conclusion?

(j) *(5 points)* Describe a real-world example of where a graph data structure might be used.

Part II

Recommended Time: 60 minutes

2 Problems

2. (30 points) The *Perrin* numbers are a sequence similar to the Fibonacci numbers, where $P(n)$ defines the n^{th} Perrin number:

$$P(n) = \begin{cases} 3 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ P(n-3) + P(n-2) & \text{if } n > 2 \end{cases}$$

The first ten Perrin numbers are 3, 0, 2, 3, 2, 5, 5, 7, 10, 12

(a) *(8 points)* Design the standard recursive `int perrin(int n)` method that returns the n^{th} Perrin number. If $n < 0$, throw an `IllegalArgumentException`.

(b) *(8 points)* Design the tail recursive `int perrinTR(int n)` method that solves the same problem as (a), but with tail recursion. You will need to design a helper method. Remember the relevant access modifiers! Do *not* handle the negative case.

(c) *(8 points)* Third, design the `int perrinLoop(int n)` method that solves the problem using a loop. Do *not* handle the negative case.

(d) *(6 points)* What is the asymptotic runtime of `perrin` in the worst case? What about `perrinTR`? What about `perrinLoop`? No justification is necessary.

- `perrin`: _____
- `perrinTR`: _____
- `perrinLoop`: _____

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 implement a class hierarchy to represent and manipulate lines in 2-D space.

   First, assume that the following `Point` class exists:

```
class Point {

  private double x;
  private double y;

  Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  double getX() { return this.x; }
  double getY() { return this.y; }
  void setX(double x) { this.x = x; }
  void setY(double y) { this.y = y; }
}
```

   (a) *(4 points)* Design the `ILineSegment` interface, which represents some type of line segment. It should contain the following methods: `double length()`, `boolean contains(int x, int y)`, `boolean contains(Point p)`, and `ILineSegment translate(int dx, int dy)`.

```
______________ ILineSegment {

  /**
   * Returns the total length of the line segment.
   */
  ___________________ length();

  /**
   * Determines if a given point at (x, y) is on the line segment.
   * @param x x-coordinate.
   * @param y y-coordinate.
   * @return true if (x, y) is on the line segment and false otherwise.
   */
  ___________________ contains(___________________________);

  /**
   * Determines if a given Point object at coordinates (x, y) is on the line segment.
   * @param p Point to check.
   * @return true if p is on the line segment and false otherwise.
   */
  ___________________ contains(___________________________);

  /**
   * Performs a linear translation of this line segment.
   * @param dx x-coordinate to translate by.
   * @param dy y-coordinate to translate by.
   * @param a new line segment representing the translation of this line segment.
   */
  ___________________ translate(_______________________________);
}
```

(b) *(14 points)* Design the `AbstractLineSegment` class, which represents a line segment that contains points. It should implement the `ILineSegment` interface and override the methods by declaring them abstract. It should contain a single variadic constructor for receiving `Point` objects to an underlying (instance variable) list of points that comprise the line segment.

```
__________ __________ AbstractLineSegment __________ __________ {

  __________ final ____________________________________ POINTS;

  AbstractLineSegment(_________________________) {
    this.POINTS = new LinkedList<>(Arrays.asList(_________));
  }

  @Override
  __________ abstract _____________________ length();

  @Override
  __________ __________ _____________________ containsPoint(Point p);

  @Override
  __________ __________ _____________________ containsPoint(int x, int y);

  @Override
  __________ __________ _____________________ translate(int dx, int dy);

  @Override
  __________ __________ toString() { return this.POINTS.toString(); }

  List<Point> getPoints() { return this.POINTS; }
}
```

(c) *(21 points)* Design the `LinearLineSegment` class, which represents a line over a linear equation $y = mx + b$. It should extend the `AbstractLineSegment` class and contain two constructors: `LinearLineSegment(int x1, int y1, int x2, int y2)`, representing the starting and ending points (x_1, y_1), (x_2, y_2) respectively, and `LinearLineSegment(Point start, Point end)`, which receives `Point` objects rather than exact coordinates.

Override the `length`, `contains`, and `translate` methods accordingly. To find the length of a `LinearLineSegment`, use the distance formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

To determine if a point is on a line segment (i.e., a `LinearLineSegment`), we need to know if the point is *colinear* with the line. A point p is co-linear with a line if it is not parallel with the line but shares a slope. To do this, we first need to compute the slope m of the two line end-points P_1 and P_2:

$$m = \frac{P_2.y - P_1.y}{P_2.x - P_1.x}$$

Now, we need to find the segment's y-intercept b, which we obtain by subtracting m multiplied by $P_1.x$ from $P_1.y$:

$$b = P_1.y - m \cdot P_2.x$$

Then, plug in $P.x$ for x and see if the equation equals $P.y$. If so, it is *colinear* and we continue to the next check. Lastly, just verify that $P.x$ is between $P_1.x$ and $P_2.x$ and that $P.y$ is between $P_1.y$ and $P_2.y$.

To translate a `LinearLineSegment`, return a new `LinearLineSegment` with the coordinates offset by the provided `dx` and `dy`.

```
__________ LinearLineSegment ____________ _________________________________ {

  LinearLineSegment(Point start, Point end) {
    super(_____________________________________________________);
  }

  LinearLineSegment(double x1, double y1, double x2, double y2) {
    this(_____________________________________________________);
  }

  @Override
  __________ __________ length() {
    Point p1 = this.getPoints().getFirst();
    Point p2 = ________________________________________________;
    return ____________________________________________________________________________;
  }

  @Override
  __________ __________ containsPoint(Point p) {
    // Determine if the three points are co-linear.
    // 1. Find the slope of the two end-points of the segment.
    Point p1 = this.getPoints().getFirst();
    Point p2 = this.getPoints().getLast();
    double m = ________________________________________________;
    double b = ________________________________________________;
    // y = mx, plug p.getX() in for x, check if equal to p.getY().
    if (_______________________________ == p.getY()) {
      // Now check to see where it lies.
      return p.getX() >= Math.min(_______________________________)
              && p.getX() <= Math.max(_______________________________)
              && p.getY() >= Math.min(_______________________________)
              && p.getY() <= Math.max(_______________________________);
    } else {
      return false;
    }
  }

  @Override
  __________ __________ containsPoint(int x, int y) {
    return this.containsPoint(_____________________________);
  }

  @Override
  __________ __________ translate(int dx, int dy) {
    double newX1 = this.getPoints().getFirst().getX() + _______;
    double newY1 = ____________________________________________;
    double newX2 = ____________________________________________;
    double newY2 = this.getPoints().getLast().getY() + dy;
    return ____________________________________________________________________;
  }
}
```

(d) *(11 points)* Design the `MultiLineSegment` class, which extends `AbstractLineSegment` and represents a multi-segmented line. That is, rather than a straight line, it is the conjunction of multiple line segments together. Its constructor should receive a variadic number of `Point` objects, where the first represents the starting point of the line segment, and the last represents the ending point. (This detail doesn't *really* matter in the context of the class, but it might help you visualize things.)

The length of a `MultiLineSegment` is the sum of its inner line segments. Note that points P_1, P_2 form a line segment, as do points P_2, P_3, up to points P_{n-1}, P_n.

Determining if a point is on a `MultiLineSegment` is nothing more than determining if it is on any individual line segment that comprises the `MultiLineSegment`.

Translating a `MultiLineSegment` by a constant factor just offsets all comprising line segments by the factor.

It is helpful to have a method that creates the `LinearLineSegment` instances from the points that comprise the `MultiLineSegment`. So, we also provide a method, `List<LineSegment> createSegments()`, which generates a list of `LinearLineSegment` objects from the points of the `MultiLineSegment`. You should call this method in `length` and `containsPoint`.

THE SKELETON CODE IS ON THE NEXT PAGE.

```
____________ MultiLineSegment _____________ AbstractLineSegment {

  MultiLineSegment(____________________________) {
    super(_____________);
  }

  /**
   * Generates a list of line segments from the points that comprise the multi-line segment.
   */
  private List<LinearLineSegment> createSegments() {
    List<LinearLineSegment> segments = new LinkedList<>();
    for (int i = 0; i < this.getPoints() - 1; i++) {
      Point curr = this.getPoints().get(i);
      Point next = this.getPoints().get(i + 1);
      segments.add(new LinearLineSegment(curr, next));
    }
    return segments;
  }

  @Override
  ____________ _____________ length() {
    return this.createSegments().stream()
                                .mapToDouble(s -> _________________)
                                .sum();
  }

  @Override
  ____________ _____________ containsPoint(Point p) {
    return this.createSegments().stream()
                                ._________________(s -> s.containsPoint(p));
  }

  @Override
  public boolean containsPoint(int x, int y) {
    return this.containsPoint(new Point(x, y));
  }

  @Override
  public ILine translate(int dx, int dy) {
    return new MultiLineSegment(
      this.getPoints().stream()
                      .map(p -> ________________________________________________)
                      .toArray(Point[]::new));
  }
}
```

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) *(2 extra credit points)* Is the following statement true or false? Explain why. "Big-Omega represents the best-case runtime of an algorithm."

   (b) *(7 points)* Both of the following statements are true. Prove **ONE** of them using either the formal definition(s) or limits. Circle the one that you are proving.
   - $n \lg n = O(n^2 \lg n)$
   - $n \lg n = \Omega(\lg n)$

   (c) *(4 points)* Provide a real-world example of concurrency and parallelism.

   (d) *(4 points)* Describe what a mutex/lock is in the context of concurrency. What are they used for?

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

Scratch work

Scratch work

Scratch work