C212 Final Exam (150 points)
May 7, 2025

C212 Final Exam Rubric

Part I

Recommended Time: 60 minutes

1 Problem

1. (70 points) (a) *(7.5 pts) Rubric:*

   - Each blank `class`, `private`, `final`, `String` is +0.5. (2 pts)
   - The constructor is +1 for the parameter, +2 for the correct body. No partial points. (3 pts)
   - The accessor is +2.5. It should be named appropriately. No partial points. (2.5 pts)

```
class Vertex {
  private final String ID;

  Vertex(String id) {
    this.ID = id;
  }

  String getId() {
    return this.ID;
  }
}
```

   (b) *(5 points) Rubric:*

   - Each blank `public`, `boolean`, `Object` is worth 1/3 of a point. (1 pt)
   - The instanceof check is worth 2 points. If they wrote something vaguely similar to what is in the key, give them 1 point. (2 pts)
   - The cast is +0.5 points. The ID comparison is +1.5 points. No points are awarded for using `==`. (2 pts)

```
public boolean equals(Object o) {
  if (!(o instanceof Vertex)) {
    return false;
  } else {
    Vertex othVertex = (Vertex) o;
    return this.ID.equals(othVertex.ID);
  }
}
```

   (c) *(7.5 points) Rubric:*

   - `abstract` and `class` are each worth +0.5 points (1 pt).
   - `private`, `Vertex`, +0.25 each. (1 pt)
   - The constructor parameters are +0.25 each. (0.5 pts)
   - The instance variable assignment is +0.5 for each instance variable assigned. (1 pt)
   - Each accessor is +2 points. Names must be appropriate. No partial marks. (4 pts)

```
abstract class Edge {
  private final Vertex SOURCE;
  private final Vertex DESTINATION;

  Edge(Vertex src, Vertex dest) {
    this.SOURCE = src;
    this.DESTINATION = dest;
  }

  Vertex getSource() { return this.SOURCE; }
  Vertex getDestination() { return this.DESTINATION; }
}
```

   (d) *(5 points) Rubric:*

- Each blank `public`, `boolean`, `Object` is worth 1/3 of a point. (1 pt)
- The instanceof check is worth 2 points. If they wrote something vaguely similar to what is in the key, give them 1 point. (2 pts)
- The cast is +0.5 points. The `Vertex` objects must be compared using `.equals`. +0.75 for each one done correctly. (2 pts)

```java
public boolean equals(Object o) {
  if (!(o instanceof Edge)) {
    return false;
  } else {
    Edge othEdge = (Edge) o;
    return this.SRC.equals(othEdge.SRC)
        && this.DEST.equals(othEdge.DEST);
  }
}
```

(e) *(10 points) Rubric:*

- `extends` and `Edge` are +1 each (2 pts)
- `private` and `double` are +0.25 each. (0.5 pts). If the variable is final, do not award either of these points.
- `Vertex`, `Vertex`, and `double` parameters are +0.5 each. (1.5 pts)
- Call to `super()` is +3 points. No partial credit. The assignment of `weight` is +1 point. (3 pts)
- Accessor and mutator for `weight`. +1 each. No partial credit. (2 pts) If they rewrite the accessors for the `Vertex` variables, they do not receive the points for this part.

```java
class WeightedEdge extends Edge {
  private double weight;

  Edge(Vertex src, Vertex dest, Vertex weight) {
    super(src, dest);
    this.weight = weight;
  }

  double getWeight() { return this.weight; }
  void setWeight(double weight) { this.weight = weight; }
}
```

(f) *(5 points) Rubric:*

- Each blank `public`, `boolean`, `Object` is worth 1/3 of a point. (1 pt)
- The instanceof check is worth 1 points. If they wrote something vaguely similar to what is in the key, give them 0.5 point. (1 pts)
- The cast is +0.5 points. They must appropriately call `super.equals` and then compare the weights. The `super` call is worth +2 points. Comparing the weight is +0.5. (3 pts). If they don't call `super.equals`, no points are awarded for this entire part (so, the 3 pts are not awarded.)

```java
public boolean equals(Object o) {
  if (!(o instanceof WeightedEdge)) {
    return false;
  } else {
    WeightedEdge othWEdge = (WeightedEdge) o;
    return super.equals(othWEdge) && this.weight == othWEdge.weight;
  }
}
```

(g) *(5 points) Rubric:*

- +1 for the `interface` keyword (1 pt)
- +2 for the `Set<Vertex>`. (2 pts)
- +2 for the `Set<Edge>`. (2 pts)

```
interface IGraph {

  Set<Vertex> vertices();

  Set<Vertex> edges();
}
```

(h) *(15 points) Rubric:*

- (No points are awarded for the `class` and `implements`).
- `private` and `List<Edge>` or `List<WeightedEdge>` are +1 each (2 pts).
- Constructor instantiates the list to an `ArrayList`. (2 pts)
- `vertices()` has `public` and correct return type; +0.5 each. (1 pt)
- Definition of `vertices()` is correct; it loops over all of the edges and adds the vertices to a set, or something similar to that... (3 pts)
- `edges()` has `public` and correct return type; +0.5 each. (1 pt)
- Definition of `edges()` is correct; it loops over all of the edges and adds them to a set. (3 pts)
- The "`weight`" version of `addEdge` is correct; it adds a new `WeightedEdge` instance to the list. (1.5 pts)
- The other version of `addEdge` is correct; it calls the "`weight`" version with the appropriate parameters. (1.5 pts).

```
class WeightedGraph implements IGraph {

  private List<Edge> edges;

  WeightedGraph() {
    this.edges = new ArrayList<>();
  }

  @Override
  public Set<Vertex> vertices() {
    Set<Vertex> vertices = new HashSet<>();
    for (Edge edge : this.edges) {
      vertices.add(edge.getSource());
      vertices.add(edge.getDestination());
    }
    return vertices;
  }

  @Override
  public Set<Edge> edges() {
    Set<Edge> edgeSet = new HashSet<>();
    for (Edge edge : this.edges) {
      edgeSet.add(edge);
    }
    return edgeSet;
  }
```

```
    void addEdge(Vertex v, Vertex w, int weight) {
      this.edges.add(new WeightedEdge(v, w, 1));
    }

    void addEdge(Vertex v, Vertex w) {
      this.edges.add(new WeightedEdge(v, w, 1));
    }
  }
```

(i) *(5 points) Rubric:*

There may be multiple answers, but the primary one that I'm looking for is that it's a redundant design choice. If they have something reasonable, award points. We can just use `WeightedEdge` and use a weight of 0.

(j) *(5 points) Rubric:*

Multiple possible answers. A simple answer is something like Google Maps/GPS.

Part II

Recommended Time: 60 minutes

2 Problems

2. (30 points) (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`.

   Rubric:

   - Case when $n < 0$. Conditional is +0.75, correct exception being thrown is +1.25 *(2 pts)*
   - Case when $n == 0$. Conditional is +0.50, correct return value is +0.50. *(1 pts)*
   - Case when $n == 1$. Conditional is +0.50, correct return value is +0.50. *(1 pts)*
   - Case when $n == 2$. Conditional is +0.50, correct return value is +0.50. *(1 pts)*
   - Recursive case. One recursive call for $n - 3$ is +1, one recursive call for $n - 2$ is +1, summing the two is +1. *(3 pts)*

```java
static int perrin(int n) {
  if (n == 0) { return 3; }
  else if (n == 1) { return 0; }
  else if (n == 2) { return 2; }
  else {
    return perrin(n - 3) + perrin(n - 2);
  }
}
```

   (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.

   Rubric:

   - If the helper method is private, deduct 2 points. If it is, do nothing.
   - Case when $n == 0$. Conditional is +0.50, correct return value is +0.50. *(1 pts)*
   - Case when $n == 1$. Conditional is +0.50, correct return value is +0.50. *(1 pts)*
   - Case when $n == 2$. Conditional is +0.50, correct return value is +0.50. *(1 pts)*
   - Recursive case. Correctly decrementing n is +2. Correctly updating the accumulators is +2. Not sure how to award partial points. *(4 pts)*

```java
static int perrinTR(int n) {
  return perrinTRHelper(n, 3, 0, 2);
}

private static int perrinTRHelper(int n, int a, int b, int c) {
  if (n == 0) { return a; }
  else if (n == 1) { return b; }
  else if (n == 2) { return c; }
  else {
    return perrinTRHelper(n - 1, b, c, a + b);
  }
}
```

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

Rubric:

- Loop condition is correct. Award partial credit liberally. *(4 pts)*
- Updates variables correctly. Award partial credit liberally. *(3 pts)*
- Correct return value. *(1 pt)*

```
static int perrinLoop(int n) {
  int a = 3;
  int b = 0;
  int c = 2;
  while (!(n == 0 || n == 1 || n == 2)) {
    int tA = a;
    int tB = b;
    a = tB;
    b = c;
    c = tA + tB;
    n = n - 1;
  }

  if (n == 0) { return a; }
  else if (n == 1) { return b; }
  else { return c; }
}
```

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

Solution.

Rubric:

- Each problem is 2 points. The correct class is +1, the correct bound is +1. *(6 pts)*
- `perrin`: $\Theta(2^n)$
- `perrinTR`: $\Theta(n)$
- `perrinLoop`: $\Theta(n)$

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)`.

   Rubric:

   - Each blank is +0.5. **Note: on the actual exam, I said that the `length` method should return `int`, but this doesn't really make sense; it should return a `double`.** Award points in either case.

```
interface ILineSegment {

  /**
   * Returns the total length of the line segment.
   */
  double 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.
   */
  boolean contains(int x, int y);

  /**
   * 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.
   */
  boolean contains(Point p);

  /**
   * 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.
   * @return a new line segment.
   */
  ILineSegment translate(double dx, double dy);
}
```

(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.

Solution.

Rubric:

- +0.5 for each blank EXCEPT FOR the ones listed below. *(9 pts)*
- +1 for the `List<Point>` blank. *(1 pt)*
- +3 for the variadic arguments. *(3 pt)*
- +1 for the `pts` blank. *(1 pt)*

```
abstract class AbstractLineSegment implements ILineSegment {

  private final List<Point> POINTS;

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

  @Override
  public abstract double length();

  @Override
  public abstract boolean containsPoint(Point p);

  @Override
  public abstract boolean containsPoint(int x, int y);

  @Override
  public abstract ILineSegment translate(int dx, int dy);

  @Override
  public String 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`.

Rubric:

- +1 for `class`, `extends`. *(2 pts)*
- +1 for `AbstractLineSegment`. *(1 pts)*
- +2 for correctly passing the points as variadic arguments in the first constructor. *(2 pts)*
- +2 for correctly creating two `Point` objects and passing them to the other constructor. No other way to earn these points. *(2 pts)*
- +0.25 for each `public` and +0.25 for each return type. *(2 pts)*
- +4 for the distance formula. Not sure how to award partial points. *(4 pts)*
- +0.5 for each correct blank in `containsPoint(Point p)`. *(3.5 pts)*
- +1.5 for correct `containsPoint(int x, int y)`. *(1.5 pts)*
- +0.75 for each correct blank in `translate`.

```java
class LinearLineSegment extends AbstractLineSegment {

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

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

  @Override
  public double length() {
    Point p1 = this.getPoints().getFirst();
    Point p2 = this.getPoints().getLast();
    return Math.sqrt(Math.pow(p2.getX() - p1.getX(), 2),
                     Math.pow(p2.getY() - p1.getY(), 2));
  }

  @Override
  public boolean 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 = (p2.getY() - p1.getY()) / (p2.getX() - p1.getX());
    double b = p1.getY() - m * p2.getX();
    // y = mx, plug p.getX() in for x, check if equal to p.getY().
    if (m * p.getX() == p.getY()) {
      // Now check to see where it lies.
      return p.getX() >= Math.min(p1.getX(), p2.getX())
             && p.getX() <= Math.max(p1.getX(), p2.getX())
             && p.getY() >= Math.min(p1.getY(), p2.getY())
             && p.getY() <= Math.max(p1.getY(), p2.getY());
    } else {
      return false;
    }
  }
  @Override
  public boolean containsPoint(int x, int y) {
    return this.containsPoint(new Point(x, y));
  }

  @Override
  public ILineSegment translate(int dx, int dy) {
    double newX1 = this.getPoints().getFirst().getX() + dx;
    double newY1 = this.getPoints().getFirst().getY() + dy;
    double newX2 = this.getPoints().getLast().getX() + dx;
    double newY2 = this.getPoints().getLast().getY() + dy;
    return new LinearLineSegment(newX1, newY1, newX2, newY2);
  }
}
```

(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.

Solution.

Rubric:

- +0.5 for `class` and +0.5 for `extends`. *(1 pt)*
- +1 for correct parameter to constructor and +1 for correct call to `super`. *(2 pts).*
- +0.5 each for `public` and the return types. *(2 pts)*
- +2 for correct `length`. *(2 pts)*
- +2 for correct `containsPoint(Point p)`. *(2 pts)*
- +2 for correct `translate(int dx, int dy)`. *(2 pts)*

```java
class MultiLineSegment extends AbstractLineSegment {

  MultiLineSegment(Point ... pts) {
    super(pts);
  }

  /**
   * 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
  public double length() {
    return this.createSegments().stream().mapToDouble(s -> s.length()).sum();
  }

  @Override
  public double containsPoint(Point p) {
    return this.createSegments().stream().anyMatch(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 -> p.translate(dx, dy))
                      .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."

   Rubric:

   - No. It's a lower-bound. All or nothing points.

   (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.

   - $n \lg n = O(n^2 \lg n)$
   - $n \lg n = \Omega(\lg n)$

   Rubric:

   - Do your best to reason through these proofs. These are VERY simple because all you have to do is divide the terms and you'll see what happens... Ask me if you are unsure about a student's solution.

   First, we'll prove that $n \lg n = O(n^2 \lg n)$ with the formal definition. This means that there exists constants c, n_0 such that $n \lg n \leq c \cdot n^2 \lg n$ for all $n \geq n_0$. Dividing both sides of the equation gets us $1 \leq cn$, which is true for any constant $c \geq 1$ and $n_0 \geq 1$.

   We can also show this via a limit; the limit must not diverge to be true.

$$\lim_{n \to \infty} \frac{n \lg n}{n^2 \lg n} = \lim_{n \to \infty} \frac{1}{n} = 0$$

   Second, we'll prove that $n \lg n = \Omega(\lg n)$ with the formal definition. This means that there exists constants c, n_0 such that $n \lg n \geq c \cdot \lg n$ for all $n \geq n_0$. Dividing both sides of the equation gets us $\lg n \geq c$. There is always a c smaller than the infinitely growing n.

   We can also show this via a limit; the limit must not be zero to hold true.

$$\lim_{n \to \infty} \frac{n \lg n}{\lg n} = \lim_{n \to \infty} n = \infty$$

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

   Rubric:

   - There are many, but a concurrent example is having a kiosk with two queues served by one cashier. They poll, e.g., from queue A, then from queue B, thereby never starving either queue.
   - A parallelism example is having a kiosk with multiple cashiers and multiple queues, processing people from both queues simultaneously.

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

   Rubric:

   - A mutex/lock is an operator/operation under-the-hood that prevents threads from accessing the same piece of data at the same time as another thread. They are used to prevent race conditions.

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

   Rubric:

   - Answers vary.