C212 Final Exam (150 points)
December 16, 2024

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 25 extra credit points for a total possible score of 175/150.

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: _____

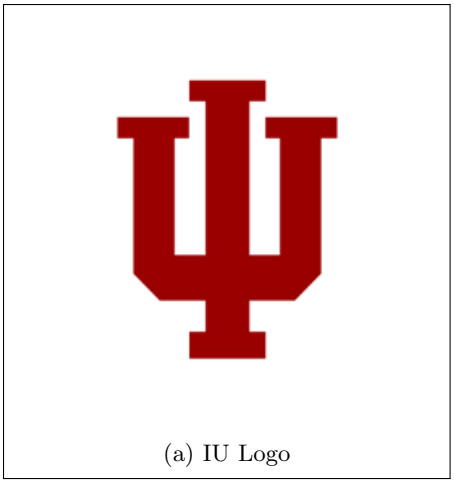
IU Email: _____

Part I

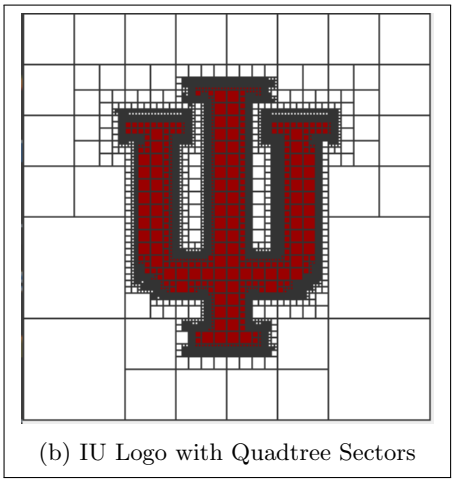
Recommended Time: 60 minutes

1 Problem

1. (70 points) A *quadtree* is a recursive data structure used to compress data efficiently. Consider an image where each pixel is represented by a color value. For instance, storing individual pixels of the IU logo that is 256×256 requires 262,144 bytes (or 262K). While this may seem negligible with today's storage capacities, scaling the image to 2048×2048 increases the required space to 16MB, which can become costly when we need to store a lot of images of this size. Quadtrees help in compressing these images by consolidating large regions of the same color.



(a) IU Logo



(b) IU Logo with Quadtree Sectors

First, assume that we have the following `Image` class:

```
class Image {

  private final int WIDTH;
  private final int HEIGHT;
  private final Color[][] PIXELS;

  Image(int width, int height, Color[][] pixels) {
    this.WIDTH = width;
    this.HEIGHT = height;
    for (int i = 0; i < this.HEIGHT; i++) {
      for (int j = 0; j < this.WIDTH; j++) {
        this.PIXELS[i][j] = pixels[i][j];
      }
    }
  }

  Color getPixel(int x, int y) { return this.PIXELS[x][y]; }

  int getWidth() { return this.WIDTH; }

  int getHeight() { return this.HEIGHT; }
}
```

You are tasked with designing the `QuadTree` class, which builds a `QuadTree` from an `Image`. We will design the class incrementally.

Quadtrees are made up of *sectors*. A `QuadTree`, in fact, is nothing more than four sectors (hence the prefix "quad"). A sector contains an (x, y) coordinate pair denoting the center coordinates of the sector, and a size s of the sector. For our purposes, all images will be square, with dimensions that are divisible by 4, so all sectors are also square-shaped.

(a) *(15 points)* Design the `Sector` class, whose constructor receives an x, y, and s denoting the center (x, y) coordinate pair and the dimensions of the sector respectively. The class also stores four separate Sector instance variables representing the `topLeft`, `topRight`, `bottomLeft`, and `bottomRight` "sub-divisions" of the `QuadTree`. In the constructor, initialize these fields to `null`, but design setter methods for them. You should also design the necessary accessor methods.
The skeleton code is below.

```
class Sector {

  private final int CENTER_X; // Center X.
  private final int CENTER_Y; // Center Y.
  private final int SIZE;     // Size of the sector.

  private Sector topLeft;
  private Sector topRight;
  private Sector bottomLeft;
  private Sector bottomRight;

  Sector(____________________________________) {









  }

  ________ getTopLeft() { return ________________; }

  ________ getTopRight() { return ________________; }

  ________ getBottomLeft() { return ________________; }

  ________ getBottomRight() { return ________________; }

  _____ setTopLeft(___________) { ________________________________________; }

  _____ setTopRight(___________) { ________________________________________; }

  _____ setBottomLeft(___________) { ________________________________________; }

  _____ setBottomRight(___________) { ________________________________________; }

  _____ getCenterX() { return this.CENTER_X; }

  _____ getCenterY() { return this.CENTER_X; }

  _____ getSize() { return this.SIZE; }
}
```

(b) *(10 points)* Design the `QuadTree` class instance variables and constructor. Its constructor receives an `Image` and assigns it to an instance variable. The constructor should also instantiate a `Sector` instance variable representing the root of the quadtree. *Hint: what is the "center" of an image? Those are the coordinates to pass to the `Sector` constructor.*

```
class QuadTree {



  QuadTree(_______________) {




  }
}
```

(c) *(15 points)* Now, we get to the heart of the quadtree: sub-divisions. The quadtree is populated based on the following rules:

(i) If a sector s contains only pixels of the same color, it should not be further sub-divided.

(ii) Otherwise, split s into four smaller sub-sectors, representing the top-left, top-right, bottom-left, and bottom-right "squares." Recursively sub-divide these sectors.

So, as you can most likely guess (from the fact that we literally stated it one sentence ago), `subdivide` is recursive, but interestingly, it does not return a value! Instead, it receives a `Sector` s and sets its fields accordingly (remember those setter methods that you defined in the last question?). But, before we design `subdivide` we need to figure out what it means for a sector to "contain only pixels of the same color." Of course, exactly as it sounds, it means we need to traverse over the pixels that this sector represents and check to see if they are all the same color.

Design the `private static boolean isUniformSector(Sector s, Image I)` method that returns whether the pixels of I that sector s represents are all the same color. You can compare two colors using the `.equals` implementation of `Color`. *Hint: use two `for` loops, and make sure that you understand what are the lower and upper bounds for the loops. You want to traverse from the top-left to the bottom-right of the sector, and you know what the center coordinate is, as well as the sector size. Do the math!*

The skeleton code is on the next page.

```
class QuadTree {

  // ... other information not shown.

  /**
   * A sector is uniform with respect to an image if all pixels in the
   * sector are of the same color.
   * @param s the sector to check.
   * @param img the image whose pixels to check.
   * @return true if the sector is uniform, false otherwise.
   */
  private static boolean isUniformSector(Sector s, Image img) {
    Color c = null;
    int startX = sector.getX() - sector.getSize() / 2;
    int endX = ________________________________________;
    int startY = ________________________________________;
    int endY = ________________________________________;

    // Loop over the sector pixels and check for uniformity.
    for (int x = ________; x < __________; x++) {
      for (_____________________________________________) {









      }
    }
    return ______;
  }
}
```

(d) (*20 points*) Design the `private void subdivide(Sector s)` method, which sub-divides a given sector s into four sectors if it is not uniform. So, this method should be a simple case analysis of whether s is uniform according to the method that you just wrote. There is one caveat: if the size of s is less than or equal to 1, then it cannot be further sub-divided. You *must* correctly call `isUniformSector` to receive full points. You may assume that its implementation is correct even without having fully completed part (c). *Hint: this question may be worth a lot of points, but it is extremely straightforward; do not over-complicate it!*

The skeleton code is on the next page.

```
class QuadTree {

  private static boolean isUniformSector(Sector s, Image img) { ... }


  /**
   * Subdivides the quadtree sector if necessary. We try each sector and, if it isn't
   * a "uniform" sector with respect to the image, then we subdivide it. If the sector
   * is too small, i.e., has a dimension of 1, we cannot further subdivide.
   * @param s the sector to subdivide, if necessary.
   */
  private void subdivide(Sector s) {
    if (s.getSize() == 1) {
      return;
    } else if (_____________________________________){
      int dim = s.getSize() / 2; // Use this variable if you need it.
      Sector topLeft = ___________________________________________;
      Sector topRight = ___________________________________________;
      Sector bottomLeft = ___________________________________________;
      Sector bottomRight = ___________________________________________;

      // Assign to the instance variables.









      // Recurse on each sector.









  }
}
```

Answer the following questions with at most 2-3 sentences. *Do not throw everything and the kitchen sink into your answer!*

(e) *(3 points)* In the best, average, and worst cases, what is the asymptotic runtime of `isUniformSector`? If you express your answer in terms of n, then you may assume that n is the dimension of the sector. You do not need to formally prove your answer or state any reasoning, but you must give your answer in terms of O, Ω, or Θ notation. Full points are awarded to the best choice.

(f) *(3 points)* What is an example of a "worst-case" input for a quadtree when compressing an image?

(g) *(4 points)* Consider the worst-case runtime of `subdivide`, which is an image where every pixel is a different color. Each time we subdivide, we invoke `isUniformSector`, but in the worst-case, each subdivision is exactly 1×1. Importantly, **each subdivision halves the problem size, which relates to the height of the quadtree h.**

More generally, at each level i of the quadtree, there are 4^i sectors, each of size $(n/2^i) \times (n/2^i)$. The number of pixels in each sector, therefore, is $(n/2^i)^2$.

To analyze the "amount of work done" at level i, multiply together 4^i and $(n/2^i)^2$, giving us $T'(n)$.

Then, to compute the worst-case runtime, we multiply the height of the quadtree h with T'.

Finish the derivation to compute the worst-case runtime of the `subdivide` method, $T(n)$. Give your answer in terms of $\Theta(\cdot)$. You will need to figure out what exactly h is in terms of n, but we gave you a hint in this problem—look for it! *Hint: the tight bound is one that we have not **explicitly** seen an example of in class.*

$$\begin{aligned} T(n) &= h \cdot T'(n) \\ &= \text{\_\_\_\_\_} \\ &= \text{\_\_\_\_\_} \\ &= \text{\_\_\_\_\_} \\ &= \Theta(\text{\_\_\_\_\_}) \end{aligned}$$

Part II

Recommended Time: 60 minutes

2 Problems

2. (30 points) This question has three parts.

   Consider the problem of removing adjacent characters in a string by propagation.

   For example, the string `"abba"` has two adjacent characters `"bb"`. Upon removing those, we have the string `"aa"`, which are also adjacent. Upon removing those, we have the string `""`.

   Another example is `"aabcdefff"`. We first remove the adjacent `"aa"` to get `"bcdefff"`. Then, we remove `"ff"` to get `"bcdef"`. None of the remaining characters are adjacent.

   A final example is `"bcddeddddf"`. we first remove the adjacent `"dd"` to get `"bceddddf"`. We then remove the second `"dd"` to get `"bceddf"`. Finally, we remove the third `"dd"` to get `"bcef"`.

   (a) *(6 points)* Design the `int returnAdjCharsIdx(String s)` method that, when given a string s, returns the index of the first occurrence of adjacent characters in s. If no characters are adjacent in s, return -1. Your solution can use either recursion or a loop.

   ```
   static int returnAdjCharsIdx(String s) {









   }
   ```

   (b) *(12 points)* Design the *tail recursive* `String removeAdjCharsTR(String s)` method that removes any and all adjacent characters in a given string using the process from above. Assume that `returnAdjCharsIdx` works correctly, regardless of what you wrote in part (a). You must use `returnAdjCharsIdx` in your solution to receive full credit.

   ```
   static String removeAdjCharsTR(String s) {









   }
   ```

(c) *(12 points)* Design the `String removeAdjCharsLoop(String s)` method that solves the problem using a loop. Assume that `returnAdjCharsIdx` works correctly, regardless of what you wrote in part (a). You must use `returnAdjCharsIdx` in your solution to receive full credit.

```
static String removeAdjCharsLoop(String s) {









}
```

3. (50 points) In this question, you will implement a simple payment type hierarchy for a point-of-sale system.

   (a) *(3 points)* First, design the `IPaymentMethod` interface, which contains three methods: `String paymentDetails()`, `double process(double subtotal)`, and `String paymentType()`.

```
__________ IPaymentMethod {

  /**
   * Returns the details associated with this payment. For this object
   * hierarchy, it will be the email address of the payment recipient.
   * @return String of details.
   */
   --------- ---------------------------------;

  /**
   * Processes a payment according to some specification.
   * @param subtotal total of transaction before any processing.
   * @return total amount after applying processing.
   */
   --------- ---------------------------------;

  /**
   * Returns the "kind" of payment this is.
   * @return payment type as a string.
   */
   --------- ---------------------------------;
}
```

   (b) *(3 points)* Next, design the `ITaxable` interface, which contains only one method: `double tax(double subtotal)`. This interface describes any kind of item that can be taxed, when given a subtotal.

```
__________ ITaxable {

  /**
   * Returns the amount after applying some form of a tax to the subtotal.
   * @param subtotal total before applying the tax.
   * @return taxed amount plus subtotal.
   */
  ---------- ----------------------------------;
}
```

(c) *(6 points)* Our payment hierarchy will have some classes that can throw exceptions if a transaction attempts to exceed a limit. Therefore, you will create a custom exception type.

Design the `LimitExceededException` class, which extends `RuntimeException`, whose constructor receives the "class type" as a string, the limit amount and transaction amount both as `double` values. You should pass to the superclass constructor a message of the form:

`paymentType: transaction of transactionAmount exceeds limit of limitAmount.`

```
__________ LimitExceededException _______ ___________________ {

  LimitExceededException(_____________________________________________________) {
    super(_____________________________________________________________________);
  }
}
```

(d) *(14 points)* Design the `DigitalPayment` abstract class, which implements both `IPaymentMethod` and `ITaxable`. Its constructor should receive the email of the transaction recipient and a value to represent a "convenience fee" for a digital transaction. Store these as instance variables, and design the appropriate accessor and mutator methods.

Inside the constructor, if the supplied email is either `null` or the empty string, throw an `IllegalArgumentException` with a sensible error message.

Override the four methods from the interface as follows: make `process`, `tax`, and `paymentType` abstract. Do not make `paymentDetails` abstract; instead, return the email associated with the transaction.

Finally, design the `void validateTransaction(double limitAmount, double transactionAmount)` method that, when given a limit amount and a transaction amount, if the latter is greater than the former, throw a `LimitExceededException` and pass the payment type by calling `paymentType()`, `limitAmount`, and `transactionAmount`.

The skeleton code is on the next page.

```
__________ _________ DigitalPayment __________ ____________________________________ {




  DigitalPayment(_____________________________________________) {




  }

  @Override
  _________ _________ _________ process(_____________);

  @Override
  _________ _________ _________ tax(_____________);

  @Override
  _________ _________ _________ paymentType();

  @Override
  _________ _________ paymentDetails() {


  }

  /**
   * Validates if the transaction limit is exceeded.
   * If it is, an exception is thrown.
   * @param limitAmount transaction limit.
   * @param amount total amount of transaction.
   */
  void validateTransactionLimit(double limitAmount, double amount) {







  }

  // Write the remaining getters and setters.









}
```

(e) *(12 points)* Design the `CreditCard` class, which extends `DigitalPayment`, and receives the email of its recipient as an argument to its constructor. `CreditCard` charges a flat convenience fee of $4.50 to every transaction. Its tax rate is 3.75%. Finally, its transaction limit is $1500. Store all three of these values as private and static constants.

To `tax` a `CreditCard` transaction, multiply the subtotal by the tax constant as defined above. The returned value is the amount *after* applying the tax percentage.

To `process` a `CreditCard` transaction, add the convenience fee to the taxed subtotal. Then, you should validate the transaction amount by invoking `validateTransactionLimit`. Return the transaction amount.

The `paymentType` of `CreditCard` is `"CreditCard"`.

```
class CreditCard extends ____________________ {

  _________ _________ ________ ________ CONVENIENCE_FEE = ______;
  _________ _________ ________ ________ TAX = ______;
  _________ _________ ________ ________ LIMIT = ______;

  CreditCard(String email) {
    super(___________________________________);
  }

  @Override
  public double process(double subtotal) {




  }

  @Override
  public double tax(double subtotal) {


  }

  @Override
  public String paymentType() {


  }
}
```

(f) *(12 points)* Design the `Cash` class, which implements `IPaymentMethod` and `ITaxable` but does *not* extend `DigitalPayment`. The constructor should receive the name of the recipient and store it as an instance variable. The tax rate of `Cash` payments is 5%. There is no convenience fee and no transaction limit. There is, however, a discount applied to all `Cash` transactions, which is a flat 10%.

To `tax` a `Cash` payment, multiply the subtotal by the tax percentage. The returned value is the amount *after* applying the tax percentage.

To `process` a `Cash` payment, apply the tax to the discounted subtotal, and return that value.

The `paymentType` of `Cash` is `"Cash"`.

The `paymentDetails` of `Cash` is simply the name of the recipient.

```
class Cash implements IPaymentMethod, ITaxable {

  private static final double DISCOUNT = __________;
  private static final double TAX = 1.05;

  private String recipient;

  Cash(String recipient) {

  }

  @Override
  public double process(double subtotal) {




  }

  @Override
  public double tax(double subtotal) {


  }

  @Override
  public String paymentType() {


  }
}
```

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-Oh represents the worst-case runtime of an algorithm."

   (b) *(3 extra credit points)* What is an example of a "best-case" input for a quadtree when compressing an image?

   (c) *(5 extra credit points)* Using your answer for $T(n)$ from Question 1, part (g), prove **ONE** of the following statements using either the formal definition(s) or limits. Circle the one that you are proving.
   - $T(n) = O(n^3)$
   - $T(n) = \Omega(n^2)$

   (d) *(4 extra credit points)* Design an algorithm that computes how efficient a quadtree compression is from the raw pixel data of an image. You can do this using either pseudocode, Java code, or even a mathematical analysis. In any case, you should compute the percent change from the raw data to the quadtree compression. There is not necessarily a right answer that we're looking for, so a reasonable attempt, even if incorrect, can earn *some* points!

(e) *(4 extra credit points)* What is the lower-bound for comparison-based sorting algorithms? Explain, intuitively, why this is the case. If you are capable of reproducing the proof, that is fine, but make it concise.

(f) *(2 extra credit points)* What is the difference between parallelism and concurrency?

(g) *(2 extra credit points)* What is a race condition, and how can we prevent them?

(h) *(3 extra credit points)* What's the most important thing that you learned from C212 this semester?

Scratch work

Scratch work

Scratch work