C212 Final Exam (150 points)
December 16, 2024

C212 Final Exam Rubric

Part I

Recommended Time: 60 minutes

1 Problem

1. (70 points) (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() {

  }

  ________ getTopRight() {

  }

  ________ getBottomLeft() {

  }

  ________ getBottomRight() {

  }

  ______ setTopLeft(___________) {

  }

  ______ setTopRight(___________) {

  }

  ______ setBottomLeft(___________) {

  }

  ______ setBottomRight(___________) {

  }
}
```

Solution.

Rubric:

- +3 points for each parameter to the constructor.
- +4 points for correctly assigning all instance variables. -1 point for each missing down to 0.
- +4 points for each getter. -0.5 for each incorrect part (either wrong return type or body).
- +4 points for each setter. -0.25 for wrong return type, -0.25 for wrong parameter/no parameter, +0.5 for the correct body.

```
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, topRight, bottomLeft, bottomRight;

  Sector(int cx, int cy, int size) {
    this.CENTER_X = cx;
    this.CENTER_Y = cy;
    this.SIZE = size;
    this.topLeft = this.topRight = this.bottomLeft = this.bottomRight = null;
  }

  int getX() { return this.CENTER_X; }

  int getY() { return this.CENTER_Y; }

  int getSize() { return this.SIZE; }

  Sector getTopLeft() { return this.topLeft; }
  void setTopLeft(Sector topLeft) { this.topLeft = topLeft; }

  Sector getTopRight() { return this.topRight; }
  void setTopRight(Sector topRight) { this.topRight = topRight; }

  Sector getBottomLeft() { return this.bottomLeft; }
  void setBottomLeft(Sector bottomLeft) { this.bottomLeft = bottomLeft; }

  Sector getBottomRight() { return this.bottomRight; }
  void setBottomRight(Sector bottomRight) { this.bottomRight = bottomRight; }
}
```

(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.*

Solution.

Rubric:

- +2 points for each instance variable.
- +1 for the correct parameter to constructor.
- +1 point for assigning the parameter to the instance variable.
- +4 for correctly creating a new `Sector`. Take off -1 point for each incorrect parameter. This means that they get 1 point for just instantiating the object.

```
class QuadTree {

  private final Image IMAGE;
  private final Sector ROOT;

  QuadTree(Image img) {
    this.IMAGE = img;
    this.ROOT = new Sector(this.IMAGE.getWidth() / 2,
                           this.IMAGE.getHeight() / 2,
                           this.IMAGE.getWidth());
  }
}
```

(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 ______;
  }
}
```

Solution.

Rubric:

- +1 each for correctly computing `endX`, `startY`, and `endY`.
- +1 each for the outer loop blanks.
- +3 for the correct inner loop, one for each component.
- +6 for the correct inner loop body. I'm not sure how to award partial credit... just be consistent. The main thing is having an immediate "break out" if the colors are not equal or some type of flag to designate that.
- +1 for returning `true` at the end.

```
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 sector 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 sector, BufferedImage img) {
```

```
  Color c = null;
  int startX = sector.getX() - sector.getSize() / 2;
  int endX = sector.getX() + sector.getSize() / 2;
  int startY = sector.getY() - sector.getSize() / 2;
  int endY = sector.getY() + sector.getSize() / 2;
  for (int x = startX; x < endX; x++) {
    for (int y = startY; y < endY; y++) {
      Color curr = new Color(img.getRGB(x, y));
      if (c == null) {
        c = curr;
      } else if (!curr.equals(c)) {
        return false;
      }
    }
  }
  return true;
}
```

(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 (____________________________________){
      Sector topLeft = _____________________________________________;
      Sector topRight = _____________________________________________;
      Sector bottomLeft = _____________________________________________;
      Sector bottomRight = _____________________________________________;

      // Assign to the instance variables.









      // Recurse on each sector.









  }
}
```

Solution.

Rubric:

- +4 points for the condition. This is all or nothing.
- +2 for each `Sector` declaration. Not sure how to award partial credit.
- +1 for each setter.
- +1 for each recursive call. Do not award points if they attempt to assign the recursive call to a variable; it's `void`!

```
private void subdivide(Sector s) {
  if (s.getSize() == 1) {
    return;
  } else if (!isUniformSector(s, this.IMAGE)) {
    int dim = s.getSize() / 2;
    Sector tl = new Sector(s.getX() - dim / 2, s.getY() - dim / 2, dim);
    Sector tr = new Sector(s.getX() + dim / 2, s.getY() - dim / 2, dim);
    Sector bl = new Sector(s.getX() - dim / 2, s.getY() + dim / 2, dim);
    Sector br = new Sector(s.getX() + dim / 2, s.getY() + dim / 2, dim);
```

```
      s.setTopLeft(tl);
      s.setTopRight(tr);
      s.setBottomLeft(bl);
      s.setBottomRight(br);

      this.subdivide(tl);
      this.subdivide(tr);
      this.subdivide(bl);
      this.subdivide(br);
    }
  }
```

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.

Solution.

In the best case is $\Theta(1)$. The average and worst case are $\Theta(n^2)$. $\Theta(n^2)$.

Rubric:

- -1 for use of O or Ω. No points for wrong bound.

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

Solution.

When all of the pixels are different colors.

Rubric:

- What's hilarious is that I actually gave the answer IN THE FIRST SENTENCE OF THE NEXT PROBLEM!!!

(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.*

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

$$\begin{aligned} T(n) &= h \cdot T'(n) \\ &= \lg n \cdot (4^i \cdot (n/2^i)^2) \\ &= \lg n \cdot (4^i \cdot n^2/4^i) \\ &= \lg n \cdot n^2 \\ &= \Theta(n^2 \lg n) \end{aligned}$$

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.

   Solution.

   Rubric:

   - Method works with strings that have less than 2 chars (+1.5)
   - Method returns correct index position (+3).
   - Method doesn't go out of bounds (+1).
   - Method returns `-1` if adjacent chars are not found.

```
static int returnAdjCharsIdx(String s) {
  if (s.length() <= 1) {
    return -1;
  } else {
    for (int i = 1; i < s.length(); i++) {
      if (s.charAt(i) == s.charAt(i - 1)) {
        return i - 1;
      }
    }
    return -1;
  }
}
```

   (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.

   Solution.

   Rubric:

   - Base case is correct (+3).
   - Method correctly calls `returnAdjCharsIdx` (+3).
   - Method recurses correctly (if it's not tail recursive, take off all points). (+6)

```
static String removeAdjCharsTR(String s) {
  if (returnAdjCharsIdx(s) == -1) {
    return s;
  } else {
    int adjCharsIdx = returnAdjCharsIdx(s);
    String lhs = s.substring(0, adjCharsIdx);
    String rhs = s.substring(adjCharsIdx + 2);
    return removeAdjCharsTR(lhs + rhs);
  }
}
```

(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. **Solution.**

Rubric:

- Loop condition is correct (+3).
- Method correctly calls `returnAdjCharsIdx` (+3).
- Method loop body updates correctly (+6).

```
static String removeAdjCharsLoop(String s) {
  while (!(returnAdjCharsIdx(s) == -1)) {
    int adjCharsIdx = returnAdjCharsIdx(s);
    String lhs = s.substring(0, adjCharsIdx);
    String rhs = s.substring(adjCharsIdx + 2);
    s = lhs + rhs;
  }
  return 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()`.

   Solution.

   Rubric:

   - +0.5 for `interface`.
   - +1 for `String paymentDetails()`. No partial credit.
   - +1 for `double process(double subtotal)`. No partial credit.
   - +0.5 for `String paymentType()`. No partial credit.

```
interface 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.
   */
  String paymentDetails();

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

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

   (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.

Solution.

Rubric:

- +0.5 for `interface`.
- +0.5 for `double`, +1 for `tax`, +1 for `double` parameter. No partial otherwise.

```
interface 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.
   */
  double tax(double 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.`

Solution.

Rubric:

- +1 for `class`.
- +1 for `extends`, +1 for `RuntimeException`.
- +0.5 for each parameter to the constructor. If out of order, award no points.
- +0.5 for using each parameter in a string message to the `super` call. The message doesn't need to be exactly like the one I give.

```
class LimitExceededException extends RuntimeException {

  LimitExceededException(String classType, double limitAmount, double amount) {
    super(classType + ": transaction of " + amount + " exceeds limit of " + limitAmount);
  }
}
```

(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.

Solution.

Rubric:

- +0.5 each for `abstract`, `class`, `implements`, `IPaymentMethod`, `ITaxable`.
- +1 for each instance variable. Must be `private`; no partial credit.
- +1.5 for the correct constructor. No partial points.
- +1 for each `public abstract ...` with parameters. Take off 0.25 for each missing.
- +1 for `paymentDetails`. No partial credit allowed.
- +4 for `validateTransactionLimit`. +2 for the condition, +2 for throwing the exception. No partial points.
- 0.25 for each remaining getter/setter. (no partial credit).

```
abstract class DigitalPayment implements IPaymentMethod, ITaxable {

  private String email;
  private double convenienceFee;

  DigitalPayment(String email, double convenienceFee) {
    this.email = email;
    this.convenienceFee = convenienceFee;
  }

  @Override
  public abstract double process(double subtotal);

  @Override
  public abstract double tax(double subtotal);

  @Override
  public String paymentDetails() {
    return this.email;
  }

  /**
   * 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) {
    if (amount > limitAmount) {
      throw new LimitExceededException(this.paymentType(), limitAmount, amount);
    }
  }

  double getConvenienceFee() {
    return this.convenienceFee;
  }

  void setConvenienceFee(double convenienceFee) {
    this.convenienceFee = convenienceFee;
  }

  String getEmail() {
    return this.email;
  }

  void setEmail(String email) {
    this.email = email;
  }
}
```

(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"`.

Solution.

Rubric:

- +1 for extending `DigitalPayment`.
- +0 for no correct instance variables, +1 for one correct, +2 for two correct, and +2.5 for all three correct.
- +3 for the `super` call. No partial credit.
- +2.5 for `process`.
- +2 for `tax`.
- +1 for `paymentType`.

```
class CreditCard extends DigitalPayment {

  private static final double CONVENIENCE_FEE = 4.50;
  private static final double TAX = 1.0375;

  CreditCard(String email) {
    super(email, CONVENIENCE_FEE);
  }

  @Override
  public double process(double subtotal) {
    return this.tax(subtotal) + CONVENIENCE_FEE;
  }

  @Override
  public double tax(double subtotal) {
    return TAX * subtotal;
  }

  @Override
  public String paymentType() {
    return "CreditCard";
  }
}
```

(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.

Solution.

Rubric:

- +1 for `DISCOUNT`.
- +2 for assigning the constructor parameter to the instance variable.
- +3 for `process`.
- +2 for `paymentType`.
- +2 for `paymentDetails`. *(Note: on the exam, I forgot to actually add blanks for this. Just give them the two points for free.)*
- +2 for `tax`.

```
class Cash implements IPaymentMethod, ITaxable {

  private static final double DISCOUNT = 0.90;
  private static final double TAX = 1.05;

  private String recipientName;

  Cash(String recipientName) {
    this.recipientName = recipientName;
  }

  @Override
  public double process(double subtotal) {
    return this.tax(subtotal * DISCOUNT);
  }

  @Override
  public String paymentType() {
    return "Cash";
  }

  @Override
  public String paymentDetails() {
    return this.recipientName;
  }

  @Override
  public double tax(double subtotal) {
    return subtotal * TAX;
  }
}
```

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?

   Solution.
   When all of the pixels are the same color, which means no sub-dividing occurs.
   Rubric:
   - This is basically the only answer, but I'm sure others might exist...

   (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.
   - $T(n) = O(n^3)$
   - $T(n) = \Omega(n^2)$

   Solution.
   Rubric:
   - ...?

   Proof. If $T(n) = O(n^3)$, then $T(n) \leq cn^3$ for all $n \geq n_0$. We know $T(n) = \Theta(n^2 \lg n)$, so the inequality is $n^2 \lg n \leq cn^3$. Dividing both sides by n^2 produces $\lg n \leq cn$. Let $c = 1$, which gets us $\lg n \leq n$, which is true for all $n \geq 1$. So, we choose $c = 1$ and $n_0 = 1$ and the claim is true. □

   Proof. If $T(n) = \Omega(n^2)$, then $T(n) \geq cn^2$ for all $n \geq n_0$. We know $T(n) = \Theta(n^2 \lg n)$, so the inequality is $n^2 \lg n \geq cn^2$. Dividing both sides by n^2 produces $\lg n \geq c$, which is trivially true for any $c, n_0 \in \mathbb{N}$ such that $n \geq 2^c$. So, let's choose $c = 2$ and $n = 4$ and the claim is true. □

(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!

A possible solution is to traverse through each sector until we hit a leaf, then compute the size of the sector. Use four bytes for the color, four bytes each for the (x, y), then four bytes for the size. So, each sector, if stored in a file, is sixteen bytes. I'm not sure what else students could do, but it's mainly an exercise in creativity...

```
private double ratio() {
  double v1 = this.IMG.getWidth() * this.IMG.getHeight() * 4;
  // % change: (v2-v1)/|v1| * 100
  double v2 = this.ratioHelper(this.ROOT);
  return (v2 - v1) / v1 * 100.0;
}

private double ratioHelper(Sector s) {
  // If any of the subsectors are null then it's a leaf.
  if (s.isLeaf()) {
    return 16;
  } else {
    return this.ratioHelper(s.getTopLeft())
         + this.ratioHelper(s.getTopRight())
         + this.ratioHelper(s.getBottomLeft())
         + this.ratioHelper(s.getBottomRight());
  }
}
```

(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.

Solution.

Rubric:

- +2 points for $\Omega(n \lg n)$. It MUST be exactly this bound.
- +2 points for a rough idea of why this is the case. Somewhere they should mention a decision tree and that there are $n!$ possible permutations.

The lower-bound is $\Omega(n \lg n)$. For n elements, there are $n!$ possible orderings. We create a decision tree to compare elements. At each choice we halve the problem size, meaning it's a base-two logarithmic relationship. We end up getting $\lg n!$, which by Stirling's approximation is $\Omega(n \lg n)$.

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

Solution.

Parallelism means simultaneous processing, concurrency means processing of multiple tasks, but not simultaneously. Complete a little bit of task A, then a little bit of task B, then a little bit

of task C, then back to A, ..., and so on. Parallelism would involve having multiple processors completing those tasks at the exact same time.

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

- Do your best to just award points if they get anywhere close to what a race condition is. My solution is VERY detailed.

A race condition is when two threads attempt to access shared data "at the same time." Consider two threads evaluating the following loop:

```
int x = 0;
while(true)
  x = x + 1;
```

Thread A could update x to be 1, but while it is completing the $x + 1$ operation (i.e., before the assignment), Thread B also performs the $x + 1$ operation, overwrites x, then performs the assignment *again*. So, according to B, x is now 2, but according to A, it is 1. We can prevent race conditions via mutexes, which prevent concurrent (thread) access to data.

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

Scratch work

Scratch work

Scratch work