1 Class Features

1. What is an obligatory method?

   A. a method that can be invoked using any object reference.

   B. a method that the client must invoke, such as a constructor.

   C. a method that the implementer must override.

   D. a method that the implementer must not override.

2. What is a class invariant?

   A. a condition that the method promises will be true immediately after the method finishes running

   B. a condition that the method must check at the beginning of the method

   C. a condition that the user (client) must ensure is true before calling the method

   D. a condition that is always true immediately after a constructor or method finishes running

3. Which statement is true for a `static` attribute (or field) of a class `X`?

   A. The class `X` has one copy of the attribute for every `x` object.

   B. The class `X` has the only copy of the attribute.

   C. Each `X` object has its own copy of the attribute.

   D. Only objects of classes that extend `X` have a copy of the attribute.

4. Suppose that an implementer creates a class that defines no constructors in the class body. Which one of the following statements is correct?

   A. the class must be a utility class.

   B. the class does not compile.

   C. the class compiles but is not useable.

   D. the compiler adds a default (no-argument) constructor.

5. Which of the following is required for objects of a class to be considered immutable?

   A. the class should be declared as final.

   B. the class should have no public mutators.

   C. all fields in the class should be private.

   D. class methods should not return references to class fields.

   E. all of the above.

   F. some of the above.

6. The class `MyBoolean` has a single attribute named `value` of type `boolean`. Consider the following implementation of the `equals` method:

```
public boolean equals(Object object) {

  boolean equals;
  if (object != null && this.getClass() == object.getClass()) {
        MyBoolean other = (MyBoolean) object;
        equals = this.value && other.value;
  }
  else {
        equals = false;
  }
  return equals;

}
```

   Which of the following properties does the above `equals` method **NOT** satisfy? For all non-null references `x`, `y` and `z`,

   A. `x.equals(y)` returns true if and only if `y.equals(x)` returns true

   B. `x.equals(null)` returns false

   C. `x.equals(x)` returns true

   D. if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` returns true.

7. Consider the following code fragment:

```
/**
 * This method does something
 *
 * @param i an integer
 * @pre. i>=0
 */

public void method(int i) {

  /* method body not shown */

}
```

   Which of the following statements is correct?

   A. In the method body, the implementer has to throw an exception when `i` is smaller than zero.

   B. In the method body, the implementer has to take the absolute value of `i` to ensure that its value is greater than or equal to zero.

   C. In the method body, the implementer has to set `i` to zero whenever the value of `i` is smaller than zero.

   D. None of the above.

8. What are the parts of a method signature?

    A. method name and return type

    B. method name and list of parameter types

    C. method name, list of parameter types, and return type

    D. method name, modifiers, list of parameter types, and return type

9. What is the difference between `x.equals(y)` and `x == y` for two references `x` and `y`?

    A. `x == y` does not compile

    B. `equals` compares equality of state and == compares equality of addresses

    C. `equals` compares equality of addresses and == compares equality of state

    D. there is no difference

10. What is true about the expression `x.compareTo(y)` if `x` is an instance of a type that implements the comparable interface?

    A. `x.compareTo(y)` returns a postive integer if `x` is less than `y`

    B. `x.compareTo(y)` returns a postive integer if `x` is equal to `y`

    C. `x.compareTo(y)` returns a postive integer if `x` is greater than `y`

    D. `x.compareTo(y)` must return 0 if `x` is equal to `y`

11. Consider the following class:

```
public class A {
    public static int x = 0;
    public int y;

    public A() {
        this.y = 0;
    }
}
```

Suppose that you have a reference of type `A`:

```
A a = new A();
```

Which of the following is legal but considered bad style?

    A. `a.x = 1;`

    B. `a.y = 1`

    C. `A.x = 1;`

    D. `A.y = 1;`

2 Testing

12. What makes up a unit test for a method in Java?

    A. arguments to the method and the expected result of running the method with the arguments

    B. arguments to the method that do not satisfy the preconditions of the method

    C. a program that prints out the result of running the method

    D. a program that records the result of running the method

13. What is the main purpose of testing?

    A. to find errors in your code

    B. to make sure that every line of code is necessary

    C. to make sure that every line of code is run at least once

    D. to prove that your code is correct

14. Consider the following method that determines if a numeric grade is equivalent to a letter grade of A+:

```
/**
 * Returns true if the specified grade is equivalent to a letter grade
 * of A+. A numeric grade greater than or equal to 90 is an A+.
 *
 * @param grade a numeric grade
 * @returns true if grade is equivalent to A+ and false otherwise
 * @pre. grade is less than or equal to 100
 */
public static boolean isAPlus(int grade) {
    // IMPLEMENTATION NOT SHOWN
}
```

Which of the following values of `grade` would be considered as boundary cases for the method `isAPlus`?

    A. 89

    B. 90

    C. 100

    D. 101

    E. all of the above

    F. A and B

    G. A, B, and C

15. Consider the following class that represents a temperature in degrees Celcius or degrees Fahrenheit:

```
public class Temperature {
    private double degrees;
    private String units;    // INVARIANT: this.units is equal to "C" or "F"
    public Temperature() {
        this.degrees = 0.0;
        this.units = "C";
    }

    /**
     * Changes the units of this temperature to the specified units
     * if the specified units is equal to "C" or "F", otherwise leaves
     * the current units of this temperature unchanged.
     *
     * @param the desired units of this temperature
     */
    public void setUnits(String units) {
        if (units == "C" || units == "F") {
            this.units = units;
        }
    }

    public void getUnits() {
        return this.units;
    }
}
```

Consider the following unit test for `setUnit`:

```
@Test
public void test_setUnits() {
    String units = /* SEE TABLE BELOW */
    String expected = /* SEE TABLE BELOW */

    Temperature t = new Temperature();
    t.setUnits(units);
    assertEquals(expected, t.getUnits());
}
```

Test case	units	expected
1	"C"	"C"
2	"F"	"F"
3	"Celcius"	"C"
4	"Fahrenheit"	"C"
5	new String("C")	"C"
6	new String("F")	"F"

Which test case fails?

A. 1

B. 2

C. 3

D. 4

E. 5

F. 6 (continued on next page)

G. all of the test cases fail

H. none of the test cases fail

16. Consider the following class that represents a two-dimensional point having an x-coordinate and a y-coordinate.

```
public class Point2 {
    private int x;
    private int y;

    public Point2(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object obj) {
        // implementation not shown
    }
}
```

`equals` is implemented using the guidelines discussed in class, and two points are considered equal if their `x` components are equal and their `y` components are equal. Consider the following unit test cases for `equals`:

Test case	p	q	expected value of `p.equals(q)`
1	`null`	`new Point2(0, 0)`	`NullPointerException`
2	`new Point2(0, 0)`	`null`	`false`
3	`new Point2(1, 1)`	`p`	`true`
4	`new Point2(1, 1)`	`new Point2(1, 1)`	`true`
5	`new Point2(2, 3)`	`new String("(2, 3)")`	`false`
6	`new Point2(0, 0)`	`new Point2(1, 1)`	`false`

Which test case should not be used?

A. 1

B. 2

C. 3

D. 4

E. 5

F. 6

G. all of the choices should be used

H. none of the choices should be used

17. Suppose that you want to compute the average of all of the values that are greater than 10 in a list `x`. You implement the following method:

```
/**
 * Returns the average of all values greater than 10 in the list x.
 *
 * @param x a list
 * @return the average of all values greater than 10 in the list x
 * @pre. x.size() > 0
 */
public static double avg(List<Double> x) {
    double avg = 0.0;
    int n = 0;
    for (int i = 0; i < x.size(); i++) {
        double xi = x.get(i);
        if (xi > 10.0) {
            n = n + 1;
            avg = avg + xi;
        }
    }
    return avg / n;
}
```

After testing your code, you are convinced that it is correct and you give it to your friends to use in their project. A couple of weeks later, your friends tell you that they did poorly on their project because of your code. What went wrong?

A. you forgot to test the case where `x` has only negative values

B. you forgot to test the case where `x` has only one value

C. you forgot to test the case where `x` has only positive values

D. you forgot to test the case where `x` has values all greater than 10

E. you forgot to test the case where `x` has values all less than 10

F. you forgot to test the case where `x` has some values equal to 10

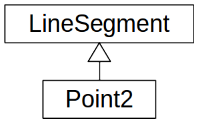
G. your friends did not use your code correctly

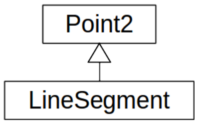
3 Aggregation and composition

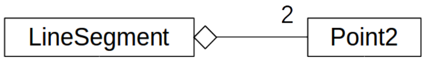
18. Consider the following class that represents a line segment connecting two points (its start point and its end point):

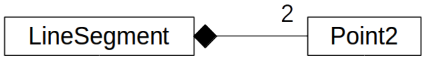
```
public class LineSegment {
    private Point2 start;
    private Point2 end;

    public Line(Point2 p1, Point2 p2) {
        this.start = p1;
        this.end = p2;
    }
}
```

Which UML diagram best describes the relationship between `LineSegment` and `Point2`?

A.


B.


C.


D.


19. In the `LineSegment` class from the previous question, the constructor contains the line:

```
this.start = p1;
```

Which choice below best completes the following statement: "`this.start` is ____________ for `p1`"

A. an alias

B. a shallow copy

C. a deep copy

D. a privacy leak

E. none of the aboe

20. When does a privacy leak occur?

A. when the client passes a reference to an object to a constructor or method

B. when an object returns the value of one of its primitive fields

C. when an object returns a reference to one of its immutable fields

D. when an object returns a reference to one of its mutable fields

21. Suppose that you want to make a shallow copy of a `List<Widget>` object where `Widget` is some class. What public feature does `Widget` need to provide for you to make a shallow copy of the list?

    A. a constructor

    B. a copy constructor

    C. a method that returns a `Widget` that is equal to another `Widget`

    D. an overridden `equals(Object obj)` method

    E. none of the above

22. Consider the following class

```
public class DeckOfCards {
    private Set<Card> cards;    /* the only field in DeckOfCards */

    public DeckOfCards() {
        /* IMPLEMENTATION NOT SHOWN but sets this.cards to be equal to
           standard deck of 52 playing cards */
    }

    public DeckOfCards(DeckOfCards other) {
        this.cards = new HashSet<>();
        for (Card card : other.cards) {
            this.cards.add(card);
        }
    }

    public Set<Card> getCards() {
        /* IMPLEMENTATION NOT SHOWN but returns either:
           an alias to this.cards OR
           a new copy of this.cards       */
    }

}
```

After the copy constructor finishes running, `this.cards` is equal to:

    A. an alias for `other.cards`

    B. a shallow copy of `other.cards`

    C. a deep copy of `other.cards`

    D. none of the above

23. Consider the following class that uses `DeckOfCard` from Question 22:

```
public class Test {
    public static void main(String[] args) {
        DeckOfCards deck = new DeckOfCards();
        DeckOfCards copy = new DeckOfCards(deck);

        Set<Card> deckCards = deck.getCards();
        Set<Card> copyCards = copy.getCards();

        System.out.println(deckCards.equals(copyCards));
    }
}
```

Which of the following is the most accurate explanation of what is printed by the program above?

A. `false` because `deckCards` contains different cards than `copyCards`

B. `false` because `deckCards` returns a reference to a different object than `copyCards`

C. `true` because `deckCards` contains the same cards as `copyCards`

D. `true` because `deckCards` returns a reference to the same object as `copyCards`

24. Consider the following class that uses `DeckOfCard` from Question 22:

```
public class Test {
    public static void main(String[] args) {
        DeckOfCards deck = new DeckOfCards();
        Set<Card> deckCards = deck.getCards();

        System.out.println(deckCards == deck.getCards());
    }
}
```

Suppose that the program above prints `true`; what is the most likely statement regarding the method `getCards`?

A. any reasonable implementation of `getCards` will always cause the program to print `true`

B. `getCards` has a privacy leak

C. `getCards` returns a shallow copy of `this.cards`

D. `getCards` returns a deep copy of `this.cards`

E. it is impossible to implement `getCards` so that the program prints `true`

4 Interfaces and Inheritance

25. What is an interface?

    A. an abstract class

    B. an abstract class that other classes may extend

    C. a class

    D. a class that other classes may extend

    E. a declaration of an API

26. Java supports single inheritance; what does the term single inheritance mean?

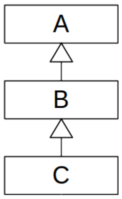
    A. every class has exactly one parent class

    B. every class has exactly one child class

    C. every class implements exactly one interface

    D. every class inherits from `Object`

27. Consider the following classes related by inheritance:



Assuming `A`, `B`, and `C` all have public no-argument constructors, which of the following variable definitions are legal?

    A. `A a = new B();`

    B. `B b = new A();`

    C. `C c = new B();`

    D. `C c = new A();`

    E. B and C

    F. B, C, and D

    G. none of the above are legal

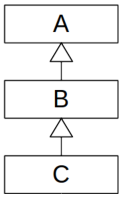
28. Consider the following classes related by inheritance:

Which of the following statements is the most accurate and complete regarding the class `C`?

    A. `C` inherits all of the fields and methods from `A`

    B. `C` inherits all of the fields and methods from `B`

    C. `C` inherits all of the fields and methods from `A` and `B` (continued on next page)



D. `C` inherits all of the fields and methods from `A` and `B` and may add its own fields and methods

E. `C` is made up of an `A` subobject

F. `C` is made up of an `B` subobject

G. `C` is made up of an `A` subobject and a `B` subobject

29. Consider the following two classes related by inheritance:

```
public class AdultCat {
    private int size;    /* INVARIANT: this.size > 0 && this.size <= 10 */

    public AdultCat(int size) {
        /* IMPLEMENTATION NOT SHOWN */
    }

    /**
     * Sets the size of this cat to be equal to sz. If sz is greater than 10
     * then the size of this cat is set to some value between 1 and 10.
     *
     * @param sz the desired size of this cat
     * @pre. sz is greater than 0
     */
    public void setSize(int sz) {
        /* IMPLEMENTATION NOT SHOWN */
    }
}

public class Lion extends AdultCat {
    public Lion(int size) {
        /* MISSING STATEMENT */
    }

    /**
     * Sets the size of this cat to be equal to sz. If sz is greater than 10
     * then /* MISSING POSTCONDITION */
     *
     * @param sz the desired size of this cat
     * @pre. /* MISSING PRECONDITION */
     */
    @Override
    public void setSize(int sz) {
        /* IMPLEMENTATION NOT SHOWN */
    }
}
```

What can the line with the comment `/* MISSING STATEMENT */` be replaced with in the `Lion` constructor?

A. `super(size);`

B. `this(size);`

C. `this.setSize(size);`

D. `this.size = size;`

30. The precondition of the `Lion` version of `setSize` is incomplete (ends with `/* MISSING PRECONDITION */`); which of the following can be used to complete the precondition so that `Lion` is substitutable for `AdultCat`?

A. `size is greater than 0 and less than 10`

B. `size is greater than 0 and less than 11`

C. `size is greater than 1`

D. `size is greater than -1`

E. `size is not equal to 0`

31. The postcondition of the `Lion` version of `setSize` is incomplete (ends with `/* MISSING POSTCONDITION */`); which of the following can be used to complete the postcondition so that `Lion` is substitutable for `AdultCat`?

A. an `IllegalArgumentException` is thrown

B. the method returns `false`

C. the size of this lion is set to `sz`

D. the size of this lion is set to `8`

32. Can the `AdultCat` constructor be safely implemented like so?:

```
public AdultCat(int size) {
    this.setSize(size);
}
```

A. no, the postcondition of `setSize` does not guarantee that the class invariant is true

B. no, `setSize` is not a `final` method

C. yes, `setSize` can be overridden by child classes

D. yes, using `setSize` minimizes code duplication

5 Complexity and Recursion

33. In plain English, what is the meaning of a big-O complexity $O(g(n))$?

    A. an estimate of best-case runtime operations of a method to solve a problem of size n

    B. an estimate of the worst-case runtime operations of a method to solve a problem of size n

    C. an estimate of the average runtime operations of a method to solve a problem of size n

    D. an estimate of the best-case memory usage for a method to solve a problem of size n

    E. an estimate of the worst-case memory usage for a method to solve a problem of size n

    F. an estimate of the average memory usage for a method to solve a problem of size n

34. If an algorithm has complexity $O(n^2)$ then which of the following statements is the most correct?

    A. doubling the size of the problem does not affect the amount of time needed for the algorithm to solve the problem

    B. doubling the size of the problem doubles the amount of time needed for the algorithm to solve the problem

    C. doubling the size of the problem increases the amount of time needed for the algorithm to solve the problem by 1 unit of time

    D. doubling the size of the problem quadruples the amount of time needed for the algorithm to solve the problem

35. Consider the following implementation of the *set* method on a singly linked-list:

```
/**
  * Sets the element stored at the given index in this linked list.
  * Returns the old element that was stored at the given index.
  *
  * @param index the index of the element to set
  * @param elem the element to store in this linked list
  * @return the old element that was stored at the given index
  * @throws IndexOutOfBoundsException if (index < 0) || (index > size)
  */
 public char set(int index, char elem) {
   if (index < 0 || index >= this.size) {
     throw new IndexOutOfBoundsException();
   }
   Node n = this.head;
   for (int i = 0; i < index; i++) {
      n = n.next;
   }
   char oldData = n.data;
   n.data = elem;
   return oldData;
}
```

What is the Big-O complexity for this method?

A. $O(1)$

B. $O(log(n))$

C. $O(n)$

D. $O(nlog(n))$

E. $O(n^2)$

36. Whenever a recursive method is called, the method:

A. always calls itself at least once

B. always calls itself zero or more times

C. always returns a value

D. never contains a loop

37. In plain English, what is the meaning of a recurrence relation $T(n)$?

A. $T(n)$ is the big-O complexity of a recursive method

B. $T(n)$ is the number of elementary operations required by the base case of the method

C. $T(n)$ is the number of elementary operations required by the recursive case of the method

D. $T(n)$ is the number of elementary operations required by the method to solve a problem of size n

38. Consider the following recursive implementation of a merge sort (pseudocode only):

```
public static LinkedList<Integer> mergeSort(LinkedList<Integer> t) {

  if (t.size() <= 1) {
    return t;
  }

  LinkedList<Integer> left = mergeSort(t.sublist(0, t.size() / 2) );
  LinkedList<Integer> right = mergeSort(t.sublist(t.size()/2 + 1, t.size() );

  LinkedList<Integer> result = merge(left,right);
  return result;
}
```

What is the recurrence relation for `mergesort` when the first base case runs?

A. $T(0) = T(1) = 0$

B. $T(0) = T(1) = c$ for some constant postive integer c

C. $T(n) = c$ for some constant postive integer c

D. $T(n) = O(n)$ where n is the number of elements in the list

39. What is the recurrence relation for `mergesort` for the recursive case where n is the number of elements in the list? Note: you may assume the `merge` method has a complexity of $O(n)$; and the value of $n/2$ is computed using integer division.

    A. $T(n) = d$ for some constant postive integer d

    B. $T(n) \approx T(n-1) + d$ for some constant postive integer d

    C. $T(n) \approx T(n/2) + d$ for some constant postive integer d

    D. $T(n) \approx 2T(n/2) + d$ for some constant postive integer d

    E. $T(n) \approx 2T(n/2) + O(n)$ for some constant postive integer d

    F. $T(n) \approx 2T(n/2) + O(n^2)$ for some constant postive integer d

40. What do you expect the Big-O complexity of `mergesort` to be?

    A. $O(1)$

    B. $O(log(n))$

    C. $O(n)$

    D. $O(nlog(n))$

    E. $O(n^2)$

6 Inheritance

41. Consider the following Java class `Shape` that represents a two-dimensional shape. Every shape has a position (the center of the shape).

```
public abstract class Shape {
        private Point2 position;
        private static int numCreated;

        public Shape(Point2 position) {
                this.position = position;
                Shape.numCreated++;
        }

        final public Point2 getPosition() {
                return this.position;
        }

        public static int getNumCreated() {
                return Shape.numCreated;
        }

        public void move(Point2 newPosition) {
                this.position = newPosition;
        }

        public abstract double getArea();
}
```

Consider the following Java class `LineSegment` that represents a two-dimensional line segment. Every line segment has a start point and an end point.

```
public class LineSegment extends Shape {
        private Point2 start;
        private Point2 end;
        private static int numCreated;

        public LineSegment(Point2 start, Point2 end) {
                /* CODE HERE NOT SHOWN; DOES NOT AFFECT LineSegment.numCreated */
                LineSegment.numCreated++;
        }

        public static int getNumCreated() {
                return LineSegment.numCreated;
        }

        public double getLength() {
                /* COMPUTES AND RETURNS THE LENGTH OF THE LINE SEGMENT HERE */
        }

        @Override
        public void move(Point2 newPosition) { /* NOT SHOWN */ }

        @Override
        public double getArea() { return 0.0; }
}
```

Finally, consider the following Java class `Circle` that represents a two-dimensional circle. Every circle has a radius.

```
public class Circle extends Shape {

        private double radius;

        public Circle(double radius, Point2 centerPoint) {
                /* SOME CODE HERE NOT SHOWN */
        }

        public final double getRadius() {
                return this.radius;
        }

        @Override
        public double getArea() { return Math.PI * this.radius * this.radius; }
}
```

(a) (3 points) Consider the following statement:

```
Shape s = new LineSegment(new Point2(0, 0), new Point2(1, 1));
```

What is the declared type (or static type) of `s`?

(b) (3 points) Consider the following statement:

```
Shape s = new LineSegment(new Point2(0, 0), new Point2(1, 1));
```

What is the actual type (or run-time type or dynamic type) of `s`?

(c) (5 points) Consider the following statements:

```
Shape s = new LineSegment(new Point2(0, 0), new Point2(1, 1));
double length = s.getLength();
```

The statement `s.getLength()` produces a compilation error. Explain how the compiler determines which methods may be called using the variable named `s`.

(d) (5 points) Suppose that you have a list of shapes named `shapes`; consider the following loop that computes the total area of all of the shapes in `shapes`:

```
/* the list shapes contains some mix of lines and circles */
double totalArea = 0.0;
for (Shape s : shapes) {
        totalArea = totalArea + s.getArea();
}
```

Explain how the Java virtual machine determines which overloaded version of `getArea` to call.

(e) (6 points) Consider the following `main` method:

```
public static void main(String[] args) {

        Shape circle1 = new Circle(1.0, new Point2(0.0, 0.0);
        Shape circle2 = new Circle(1.0, new Point2(5.0, 3.0);
        Shape circle3 = new Circle(1.0, new Point2(-8.0, -7.0);
        Shape line1 = new LineSegment(new Point2(0, 0), new Point2(1, 1));

        System.out.println("number of shapes : " + Shape.getNumCreated());
        System.out.println("number of circles: " + Circle.getNumCreated());
        System.out.println("number of lines  : " + LineSegment.getNumCreated());
}
```

Fill in the blanks below to show what the `main` method prints AND explain how you arrived at each count for the number of shapes, circles, and lines.

```
number of shapes : ______________

number of circles: ______________

number of lines  : ______________
```

(f) (3 points) The Shape method named move has the following API:

```
/**
 * Changes the position of this shape to newPosition.
 * Subclasses should override this method if they have
 * additional features that also need to be moved.
 *
 * @param newPosition the new position of this shape
 */
public void move(Point2 newPosition)
```

The LineSegment method overrides move because moving a line segment requires moving its start and end positions. The API for the LineSegment method move is:

```
/**
 * Changes the position of this line segment to newPosition.
 * Also moves the start and end point of this line segment
 * so that the line segment has the correct new position.
 *
 * @param newPosition the new position of this line
 */
@Override
public void move(Point2 newPosition)
```

Write one line of Java code to show how the LineSegment method move can set the new position of the line segment (do not try to set this.start and this.end).

7 Recursion

42. Consider the following recursive method:

```
/**
 * Returns 6n + 6, where n is the given integer.
 *
 * @param n an integer.
 * @pre. n >= 0.
 * @return 6n + 6
 */
public static int recursive(int n) {

  if (n==0) {
        return 6;
  }
  else if (n % 2 == 0) {
        return 2 * recursive(n / 2) - 6;
  }
  else {
        return recursive(n - 1) + 6;
  }
}
```

(a) (5 points) Prove that the above method `recursive` terminates.

NOTE: Not all sections discussed how to do this.

(b) (5 points) Prove that the above method `recursive` is correct.

(c) (5 points) Each of the following expressions denote the number of operations in a given method. For each expression, state the associated Big-O complexity. Formal proofs are not required.

`method 1:` $f(n) = 3n^2 + 8n + 4$ ____________________________

`method 2:` $f(n) = 8n + \frac{4}{n} + 3$ ____________________________

`method 3:` $f(n) = n^4 + 3n^2 + n$ ____________________________

`method 4:` $f(n) = nlog(n) + log(n)$ ____________________________

`method 5:` $f(n) = 10nlog(n) + 3n^2$ ____________________________

Consider the following recursive method

```
/**
 * Returns true if the character c appears in s, and false otherwise
 *
 * @param s a string
 * @param c a character to search for in s
 * @return true if the character c appears in s, and false otherwise
 */
public static boolean contains(String s, char c) {
  int result = false;
  if (s.isEmpty()) {
    result = false;
  }
  else if (s.charAt(0) == c) {
    result = true;
  }
  else {
    String t = s.substring(1, s.length());
    result = contains(t, c);
  }
  return result;
}
```

(d) (2 points) What are the base case(s) and their associated number of elementary operations for the `contains` method above.

(e) (4 points) Derive a recurrence relation for the `contains` method above, and the associated Big-O complexity. Show all working.

EXTRA BLANK PAGE

EXTRA BLANK PAGE