

**PLEASE READ ALL DIRECTIONS BEFORE STARTING YOUR EXAM.
DO NOT OPEN UNTIL YOU ARE TOLD TO DO SO.**

Permitted Materials: This is a closed-note exam except for your aid sheet. No electronic devices or communication with anyone other than proctors and the professor is allowed. *Cheating or talking with others will result in an automatic F in the course.*

Imports and Design Recipe: Unless a question states otherwise, you may use any Java class without writing its corresponding `import`. You do not need to write the full design recipe (signature, documentation, tests), though you may do so if it helps.

Disallowed Concepts: On this exam, you are only allowed to use the concepts we have talked about in class and in the textbook. Do not solve a problem using, e.g., data structures.

Errors on Exam: If you believe you have found an error on the exam, raise your hand and notify a proctor.

Upon Finishing: When you are finished, check over your work. If more than 10 minutes remain, submit your exam and aid sheet to a proctor, show your ID, then quietly exit. No one may leave during the final 10 minutes.

Removing Sheets of Paper: Do not tear out, detach, or remove any pages from this exam. If you use the scratch sheet of paper, please label the question on the sheet and put a note on the question.

Time Limit: You have **75 minutes** to complete the exam.

Good luck!

Question	Points	Score
1	20	
2	30	
3	30	
Total:	80	

Name: _____

IU Email: _____

1. (20 points) Design the double `computeFlightCost(String cabinType, String fare, int bags, int snacks, int drinks, boolean isElite)` method, which returns the cost of a flight ticket.

The `CabinType` defines the base cost of the ticket. There are three possible `CabinTypes`:

A `CabinType` is one of:

- "Economy"
- "Business"
- "First"

An "Economy" class cabin costs \$160; a "Business" class cabin costs \$280; a "First" class cabin costs \$500.

The `Fare` defines an additional percentage on top of the base cost.

A `Fare` is one of:

- "Basic"
- "Standard"
- "Flex"

The "Basic" fare adds an additional 20% to the base cost (i.e., the base cost is multiplied by 1.2); "Standard" adds an additional 50% to the base cost; "Flex" adds an additional 70% to the base cost.

The first and second bag each costs \$30. Any additional bags cost \$15. For example, a person carrying 3 bags would pay \$75 in bag fees.

Each snack costs \$3, each drink costs \$4. For a person in the "First" class cabin, however, all snacks and drinks cost \$0.

Finally, an "elite" passenger's ticket can never be more than \$1,000, even if the subtotal exceeds that amount.

In designing this method, follow the design recipe from class: write the signature, purpose statement, testing, and *then* do the implementation. You may assume that all inputs are well-formed.

For the tests, write THREE different tests that test the three `CabinTypes`. Delta values are not necessary.

The skeleton code is on the next page. You must fill in the Java documentation comment, the tests, and the method to receive full credit.

Hint: when writing test cases, we are not requiring you to simplify the expected value. So, you don't have to compute, for example, 70% of 280. If you write out the arithmetic expression that reduces to the correct value, we will mark it as correct.

```
class FlightCostTester {

    @Test
    void testComputeFlightCost() {
        assertEquals(_____,
            computeFlightCost(_____));
        assertEquals(_____,
            computeFlightCost(_____));
        assertEquals(_____,
            computeFlightCost(_____));
    }
}

class FlightCost {

    /**
     *
     *
     * @param cabinType
     * @param fare
     * @param bags
     * @param snacks
     * @param drinks
     * @param isElite
     * @return
     */
    static double computeFlightCost(String cabinType, String fare, int bags,
                                     int snacks, int drinks, boolean isElite) {

    }
}
```

2. (30 points) This question has three parts, each of which is weighted equally.
- (a) (10 points) Design the `String removeRunsOfLength3(String s)` method that, when given a string s , removes all runs of length 3 of the same character. For example, `removeRunsOfLength3("aaabcddddeef")` returns "bcdeef".
- Your method must be standard recursive or you will receive no credit.**

- (b) (10 points) Design the *tail recursive* `String removeRunsOfLength3TR(String s)` method that solves the same problem as (a), but uses tail recursion. You will need to design a helper method. Remember the relevant access modifiers!

Your helper method must be tail recursive or you will receive no credit.

- (c) (10 points) Design the *iterative* `String removeRunsOfLength3Loop(String s)` method that solves the same problem as (a) and (b), but uses a loop.

Your method must use a loop and not be recursive or you will receive no credit.

Before proceeding...

For the last question, did you use the translation pipeline to solve the problem? It was not required; I am just curious!

----- Yes

----- No

3. (30 points) Design the `String removeOccurrences(String s, String m)` method that, when given two strings s and m , removes all occurrences of m in s .

You must account for overlapping occurrences. That is, suppose you call the method with the arguments `removeOccurrences("abcturkturkeyeycba", "turkey")`. When removing the instance "turkey", we get a new string "abcturkeycba" because we combine the strings "abcturk" and "eycba". The instance of "turkey" created by that concatenation should also be removed, producing "abccba".

In designing this method, follow the design recipe from class: write the signature, purpose statement, testing, and *then* do the implementation. You may assume that all inputs are well-formed.

For the tests, you must write at least THREE tests that comprehensively cover the inputs. It is up to you to figure out what that means!

Note: Your algorithm must use either recursion or a loop; you cannot use built-in methods that do the “heavy lifting” such as `replace` or `replaceAll`.

The skeleton code is on the next page. You must fill in the Java documentation comment, the tests, and the method to receive full credit.

```
class RemoveOccurrencesTester {

    @Test
    void testRemoveOccurrences() {
        assertEquals(_____,
            removeOccurrences(_____));
        assertEquals(_____,
            removeOccurrences(_____));
        assertEquals(_____,
            removeOccurrences(_____));
    }
}

class RemoveOccurrences {

    /**
     *
     * @param s
     * @param m
     * @return
     */
    static String removeOccurrences(String s, String m) {

    }
}
```

Scratch work