Outlab 8: Basic Java

Edits: All edits are highlighted in green

Please refer to the general and submission instructions mentioned at the end of the document before submitting.

Java-Basic will have mixed grading - that is, some questions will be graded and some will be ungraded. P1, P2 and P4 are all going to be graded. P3 and P5 are ungraded, so prioritize your work accordingly!

However, we strongly recommend you to try all the problems for your own good.

RESOURCES LINK :
https://drive.google.com/drive/folders/1-1DPFqZeTf1JkxesNUP0_6Zpat3HLdpE

Problem 1 (15 Marks)

MD5.java

MD5 (Message-Digest algorithm 5) is a widely-used cryptographic hash function with a -bit hash value. Here are some common uses for MD5:

- To store a one-way hash of a password.
- To provide some assurance that a transferred file has arrived intact.

For the first task, you will be given a ***MD5sums*** file. Each line in the file has plaintext, followed by a "-", followed by its **MD5** value. We want you to parse a file, check the **plaintext** and its corresponding **MD5**, and if they match, print "***verified***". If they do not match, print "***not verified***" in a new line in the same order that you check each entry.

We will run your script as follows.

```
javac MD5.java //to compile your script.
java MD5 //to run your script.
```

Expected output :

```
verified // first input pair is correct.
not verified // second input pair is incorrect.
...
```

We will run your program with another file and compare the expected output(the file name will be the same, ***MD5sums***)

It will be auto-graded so pay attention to case-sensitivity. Small “**v**” and small “**n**” in “***verified***” and “***not verified***” respectively.

GUI.java

Design a GUI for the above ***MD5.java*** program. You can do this using “*java.swing*”. The GUI should contain two buttons, “***Select file***” and “***Process***”.

Clicking on "***Select file***" should display *JFileChooser* to select the desired file. Clicking on "***Process***" should accomplish the task ***MD5.java*** stated above, but this time you should display the results in a *JTable*, with one column as the plain-text and other as result(verified or not). There’s no need to print the MD5 value column.

Problem 2 - Pollution Control (20 Marks)

You are given the task of developing a software for the Pollution Control Board of Dunwall city. Your main task would be to generate a pollution report for each registered vehicle in the Dunwall metropolitan area.

Each vehicle in the city is either classified as a **Car** or a **Truck**. Each vehicle is identified by a **unique Registration Number** of the form of “XXX123” (3 alphabets followed by 3 digits). Each vehicle has a **manufacturer name**, and the **name of the owner.** The 2 kinds of vehicles have different thresholds for each pollutant. A vehicle must not exceed the threshold for **any** of the 3 pollutants **CO2, CO,** and **HC**.

You are given a list of vehicles in **vehicles.txt** and a list of pollution reports in **pollution.txt**. Assume that each entry in the **pollution.txt** corresponds to a **valid** vehicle in vehicles.txt. You will be queried with a vehicle registration number, and you must return whether the vehicle had passed the pollution control test. The queries are provided in **queries.txt**. The thresholds are given below

---

For Cars

CO2 <= 15 CO <= 0.5 HC <= 750

For Trucks

CO2 <= 25 CO <= 0.8 HC <= 1000

To help you get started, you have been given some guidelines:

1. Design a class **Vehicle**
    a. **Attributes (private/protected):**
        i. **String regNo:** vehicle registration number
        ii. **String manufacturer**: manufacturer name
        iii. **String owner:** name of the owner
        iv. **double co2**: CO2 emission level
        v. **double co**: CO emission level
        vi. **double hc**: HC emission level
        vii. **String pollutionStatus**: Can be “PASS”, “FAIL”, “PENDING”

    b. **Functions (public):**
        i. **void checkPollutionStatus()**: self explanatory. Hint: Modify the attribute **pollutionStatus** based on the CO2, CO and HC levels of this vehicle
        ii. **String toString():** returns a string of the form
        “Reg No: <XXX123>
        Manufacturer: <Manufacturer Name>

Owner: <Owner Name>
Pollution Status: <pollution status>

You can add other functions (public/private) and constructors of your own as you feel necessary

2. Design a class **Car** that inherits **Vehicle**:
    a. No additional derived class attributes
    b. No additional derived class methods

    Note: You should override some function of the base class to implement the correct functionality

3. Design a class **Truck** that inherits **Vehicle**:
    a. No additional derived class attributes.
    b. No additional derived class methods

    Note: You should override some function of the base class to implement the correct functionality

4. Design a class **PollutionCheck**: This is the class that contains your main method.

Input file format

vehicles.txt - each line of this file contains a string of the form
XXX123, Manufacturer Name, Owner Name, Type

pollution.txt - each line of this file contains a string of the form
XXX123, CO2, CO, HC

Queries.txt - Each line of this contains a single string of the form
XXX123

Assume that all registration IDs across all files are valid of the form XXX123.
If a vehicle in the query file is not in the vehicle list, print "NOT REGISTERED" instead.

Example vehicles.txt

WBA303, Honda, Chris Morris, Car
HRY712, Scania, Ashish Kumar, Truck
MHA569, Hyundai, Chen Wang, Car

Example pollution.txt

WBA303, 10, 0.1, 1
HRY712, 26, 0.5, 278

Example queries.txt

WBA303
HRY712
MHA569
CAC772

Example output (stdout)

PASS
FAIL
PENDING
NOT REGISTERED

The files will be given as command line arguments. Your code will be run as

java PollutionCheck <vehicles-file> <pollution-file> <queries-file>

The order of the arguments will **not** change

Problem 3 - SpaceX Mars Colonization (Ungraded)

SpaceX is ready to start a colony on Mars, and you are appointed as a chief scientist for the project. To start a civilization on the planet, SpaceX is planning to build **N** ($3 \leq N \leq 250$) cities on Mars connected by **M** ($N \leq M \leq 1000$) roads. But you need to provide electricity to each and every city.

However, SpaceX has got a limited budget and has determined that the cheapest way to arrange for electricity access is to build some overhead wires along existing roadways. You have a list of the costs of laying wires along any particular road, and want to figure out how much money you'll need to successfully complete the project – meaning that, at the end, every city will be electrified. The cost of building overhead wires along **i^{th}** road is **C_i** for **i = 1 to M**.

However, you need to set up power plants in some cities to generate electricity.
The cost of setting up a power plant at the **i^{th}** city is **A_i** for **i = 1 to N** . Any city that is connected to a city with a power plant by a sequence of overhead wires is also electrified. SpaceX ideally wants to minimize the cost of undertaking this massive project. Only you can come to the rescue now.

Design a class **ElectricityGrid** in **ElectricityGrid.java** with the following private attributes:

- *int* **noOfCities**: self explanatory
- *int* **noOfRoads**: self explanatory
- *int* **cost**: minimum cost to electrify all cities
- int[] **powerPlantCities**: An array containing the indices of the cities in which a power plant needs to be built.
- *int[][]* **gridNetwork:** An array of size **k x 2** where **k** is the number of roads along which overhead wires are to be built. **gridNetwork[i][0] and gridNetwork[i][1]** are the endpoints of the **ith** road. (This may be in any order).

The class should have the following functionalities:

- **Default constructor**
- **ElectricityGrid(int cities, int roads, int[][] wireCost, int[] powerPlantCost)**: This constructor takes **4** arguments:
  - **cities:** number of cities (**N**)
  - **roads:** number of roads (**M**)
  - **wireCost:** An array of size M x 3, the **i^{th}** entry holding an array of the **[s, d, c]** where **s and d** are 2 endpoints of the **i^{th}** road and **c** denoting the cost to lay overhead wires along that road.
- **get()** methods for all the attributes listed above. Naming convention: “get”+<attribute-name> in camelCase. Examples include:

  int getCost()
  int[] getPowerPlantCities()
  int[][] getGridNetwork()

- **void setGridPlan()**: populates the **cityWithPowerPlant** and **gridConnections and cost** attributes to hold the final plan.

NOTE: YOUR MAIN FUNCTION SHOULD BE IN A SEPARATE FILE SpaceXMain.java

We will be using the get() methods to evaluate your code.

Bonus Marks for efficient solution

Note: You are free to use any Data Structure that you may wish. You are free to implement any member functions and declare any other member variables as you wish.

Input Format: The input will be read from stdin

The first line contains **2** integers **N** and **M** representing the total number of cities and the no of roads respectively.

The next **M** lines contains three space-separated integers **x, y, c** the **i^{th}** of which denote the start, end and cost of laying wires along road **i**

The last line contains **N** space-separated integers representing cost of building a power plant in city **i**.

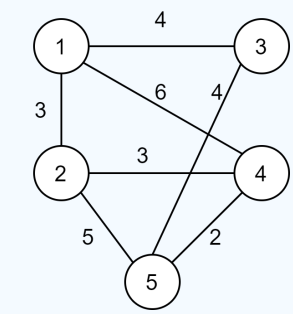
Sample input

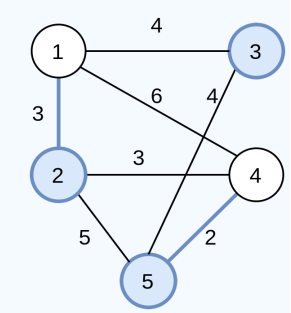
---

5 7
1 2 3
1 3 4
1 4 6
2 4 3
2 5 5
3 5 4
4 5 2
3 2 2 3 2

Solution:

Min Cost: 11





Road Network

Build power plants in cities 2, 3 and 5. Cost = 2 + 2 + 2 = 6. Connect (1,2) and (4,5)
Total cost = 2+2+2+3+2 = 11

Class attributes:

powerPlantCities = [2, 3, 5] (Order is not important)
gridNetwork = [[1, 2], [4, 5]] (Order is not important)

Sample input

---

4 4
1 2 1
1 3 1
1 4 1
2 4 1
3 2 2 3

Min cost: 5
One possible way is to set up a power plant at city 2. Build wires along (1, 2) (1, 3) (1, 4)

Problem 4 (15 Marks)

Task A - Streaming App

You have developed your own movie streaming application! Nice job, but with more and more people using your application, you may well start knowing your consumers better. You plan to know the favourite genre of each user.

You have access to the user names and for each user, a list of all the movies he/she has watched. You also have access to a list of all movies corresponding to each genre. A movie can only belong to only **one** genre.

The task is to create a map, where the key is a user name and the value is a list of the user's favorite genre(s). One's favorite genre is the most watched genre. A user can have more than one favorite genre.

Complete the function **getFavouriteGenres** in the file **StreamingApp.java**

Function name: getFavouriteGenres
Arguments:

1. **Map<String, ArrayList<String> > userMovies:** A map with username as the key and a list of all movies that the particular user has watched as values.
2. **Map<String, ArrayList<String> > movieGenres:** A map with genre as the key and a list of all movies that belong to the particular genre

Return Type: Map<String, ArrayList<String> >: Returns a map with username as key and a list of favourite genres as value

Note: It is guaranteed that the list of movies in **movieGenres** exhaustively cover all possible movies that can be shown on your platform, i.e. a user cannot watch a movie not in this list.

Sample userMovies

{

“David”: [“The Conjuring”, “Shoah”, “The Purge”, “13th”, “The Dictator”]

“Emma”: [“The Matrix”, ““Captain America: Civil War”, “John Wick”]

}

Sample movieGenres

{

“Horror”: [“The Conjuring”, “The Purge”]

“Action”: [“John Wick'']

“Documentary”: [“Shoah”, “13th”]

“Science Fiction”: [“The Matrix”, “Captain America: Civil War”]

“Comedy”: [“The Dictator”]

}

Sample return value

{

“David”: [“Horror”, “Documentary”]

“Emma”: [“Science Fiction”]

}

Task B - Sieve

Write a program which implements Sieve of Eratosthenes method for computing primes up to (and including) a specified number.

Important: Your code must accomplish this using a combination of Lambda functions, map, reduce and filter.

You can use at most one loop for 90% marks on this problem. Full marks only if you do it without any loop. For any more loops than one, no marks.

Filename: Sieve.java
Input Format: Only a single integer **N** is given as input
Output Format: Display space separated integers, with the **i^{th}** integer being the **i^{th}** prime number.

Sample input:

10

Sample output:
---
2 3 4 7

Problem 5 (Processing) (Ungraded)

Head over to https://processing.org/, It a beautiful framework & IDE based on java

It hides much of the verbosity of Java to give an environment which is especially suited for mathematicians and scientists to create animation, simulation and graphics in 2D & 3D. Explore its example library and create a mini paint application with the following features :

- Right click hold and drag the mouse to draw a line.
- Left click hold and drag the mouse to erase.

- Press ‘+’ to increase brush and eraser size
- Press ‘-’ to decrease brush and eraser size

Submission Instructions

After creating your directory, package it into a tarball **<roll-no1>-<roll-no2>-outlab8.tar.gz** in ascending order. Submit once only per team from the moodle account of the smallest roll number.

We will untar your submissions using
$ tar xvf <roll-no1>-<roll-no2>-outlab8.tar.gz
Make sure that when the above is executed, the resulting <team_name>-outlab7/ directory has the correct directory structure.

The directory structure should be as follows (nothing more nothing less, order not enforced, obviously).

<roll-no1>-<roll-no2>-outlab8/

```
├── references.txt
├── P1
│   ├── MD5sums
│   ├── MD5.java
│   └── GUI.java
├── P2
│   ├── Vehicle.java
│   ├── Car.java
│   ├── Truck.java
│   └── PollutionCheck.java
├── P3
│   ├── ElectricityGrid.java
│   └── SpaceXMain.java
├── P4
│   ├── Sieve.java
│   └── StreamingApp.java
└── P5
    └──  processing_file.java
```