Question 1-

```
/*
We are building a program to manage a gym's membership. The gym has multiple
members, each with a unique ID, name, and membership status. The program allows gym
staff to add new members, update members status, and get membership statistics.

Definitions:
* A "member" is an object that represents a gym member. It has properties for the ID,
name, and membership status.
* A "membership" is a class which is used for managing members in the gym.
*/

/*
To begin with, we present you with two tasks:
1-1) Read through and understand the code below. Please take as much time as necessary,
and feel free to run the code.
1-2) The test for Membership is not passing due to a bug in the code. Make the necessary
changes to Membership to fix the bug.
*/

/*
Gym - class
Id, name, membership status

Functions - Add,update, statistics
*/

using System;
using System.Collections.Generic;
using System.Diagnostics;

public enum MembershipStatus
{
  /*
    Membership Status is of three types: BASIC, PRO and ELITE.
    BASIC is the default membership a new member gets.
    PRO and ELITE are paid memberships for the gym.
  */
  BASIC = 1,
  PRO = 2,
  ELITE = 3
}

public class Member
```

```
{
    /* Data about a gym member. */
    public int MemberId { get; set; }
    public string Name { get; set; }
    public MembershipStatus MembershipStatus { get; set; }

    public Member(int memberId, string name, MembershipStatus membershipStatus)
    {
        MemberId = memberId;
        Name = name;
        MembershipStatus = membershipStatus;
    }

    public override string ToString()
    {
        return $"Member ID: {MemberId}, Name: {Name}, Membership Status:
{MembershipStatus}";
    }
}

public class Membership
{
    /*
        Data for managing a gym membership, and methods which staff can
        use to perform any queries or updates.
    */
    private List<Member> members;

    public Membership()
    {
        members = new List<Member>();
    }

    public void AddMember(Member member)
    {
        members.Add(member);
    }

    public void UpdateMembership(int memberId, MembershipStatus membershipStatus)
    {
        Member memberToUpdate = members.Find(member => member.MemberId ==
memberId) ?? throw new ArgumentException();
        if (memberToUpdate != null)
        {
```

```
            memberToUpdate.MembershipStatus = membershipStatus;
        }
    }

    public Dictionary<string, double> GetMembershipStatistics()
    {
        int totalMembers = members.Count;
       // Console.WriteLine(MembershipStatus.PRO);
        int totalPaidMembers = members.Where(member => member.MembershipStatus ==
MembershipStatus.PRO ||  member.MembershipStatus ==
MembershipStatus.ELITE).Count();
        // Console.WriteLine(totalPaidMembers);
        double conversionRate = (double)totalPaidMembers / totalMembers * 100;

        return new Dictionary<string, double>
        {
            { "total_members", totalMembers },
            { "total_paid_members", totalPaidMembers },
            { "conversion_rate", conversionRate }
        };
    }
}

public class TestSuite
{
    /*
        This is not a complete test suite, but tests some basic functionality of
        the code and shows how to use it.
    */
    public static void Main()
    {
        TestMember();
        TestMembership();
    }

    public static void TestMember()
    {
        Console.WriteLine("Running TestMember");
        Member testMember = new Member(1, "John Doe", MembershipStatus.BASIC);
        Debug.Assert(testMember.MemberId == 1);
        Debug.Assert(testMember.Name == "John Doe");
        Debug.Assert(testMember.MembershipStatus == MembershipStatus.BASIC);
    }
```

```
    public static void TestMembership()
    {
        Console.WriteLine("Running TestMembership");
        Membership testMembership = new Membership();
        Member testMember = new Member(1, "John Doe", MembershipStatus.BASIC);
        testMembership.AddMember(testMember);
        Debug.Assert(testMembership.GetMembershipStatistics()["total_members"] == 1);

        testMembership.UpdateMembership(1, MembershipStatus.PRO);
        Debug.Assert(testMembership.GetMembershipStatistics()["total_paid_members"] ==
1);

        Member testMember2 = new Member(2, "Alex C", MembershipStatus.BASIC);
        testMembership.AddMember(testMember2);

        Member testMember3 = new Member(3, "Marie C", MembershipStatus.ELITE);
        testMembership.AddMember(testMember3);

        Member testMember4 = new Member(4, "Joe D", MembershipStatus.PRO);
        testMembership.AddMember(testMember4);

        Dictionary<string, double> attendanceStats =
testMembership.GetMembershipStatistics();
        Debug.Assert(attendanceStats["total_members"] == 4);
        Debug.Assert(attendanceStats["total_paid_members"] == 3);
       Debug.Assert(Math.Abs(attendanceStats["conversion_rate"] - 75.00) < 0.1);
    }
}
```

Question 2- Shopping List

/*

You are going on a camping trip, but before you leave you need to buy groceries. To optimize your time spent in the store, instead of buying the items from your shopping list in order, you plan to buy everything you need from one department before moving to the next.

Given an unsorted list of products with their departments and a shopping list, return the time saved in terms of the number of department visits eliminated.

Example:

```
products = [
  ["Cheese",        "Dairy"],
  ["Carrots",       "Produce"],
  ["Potatoes",      "Produce"],
  ["Canned Tuna",   "Pantry"],
  ["Romaine Lettuce", "Produce"],
  ["Chocolate Milk",  "Dairy"],
  ["Flour",         "Pantry"],
  ["Iceberg Lettuce", "Produce"],
  ["Coffee",        "Pantry"],
  ["Pasta",         "Pantry"],
  ["Milk",          "Dairy"],
  ["Blueberries",   "Produce"],
  ["Pasta Sauce",   "Pantry"]
]
```

list1 = ["Blueberries", "Milk", "Coffee", "Flour", "Cheese", "Carrots"]

For example, buying the items from list1 in order would take 5 department visits, whereas your method would lead to only visiting 3 departments, a difference of 2 departments.

Produce(Blueberries)->Dairy(Milk)->Pantry(Coffee/Flour)->Dairy(Cheese)->Produce(Carrots) = 5 department visits
New: Produce(Blueberries/Carrots)->Pantry(Coffee/Flour)->Dairy(Milk/Cheese) = 3 department visits

list2 = ["Blueberries", "Carrots", "Coffee", "Milk", "Flour", "Cheese"] => 2
list3 = ["Blueberries", "Carrots", "Romaine Lettuce", "Iceberg Lettuce"] => 0
list4 = ["Milk", "Flour", "Chocolate Milk", "Pasta Sauce"] => 2
list5 = ["Cheese", "Potatoes", "Blueberries", "Canned Tuna"] => 0

All Test Cases:
shopping(products, list1) => 2
shopping(products, list2) => 2

shopping(products, list3) => 0
shopping(products, list4) => 2
shopping(products, list5) => 0

Complexity Variable:
n: number of products
*/

Question 3 – Stock Price comparison

```
/*
We are developing a stock trading data management software that tracks the prices of different stocks over time and provides useful statistics.

The program includes three classes: `Stock`, `PriceRecord`, and `StockCollection`.

Classes:
* The `Stock` class represents data about a specific stock.
* The `PriceRecord` class holds information about a single price record for a stock.
* The `StockCollection` class manages a collection of price records for a particular stock and provides methods to retrieve useful statistics about the stock's prices.
*/


/*
2) We want to add a new function called "GetBiggestChange" to the StockCollection class. This function calculates and returns the largest absolute change in stock price between any two consecutive days in the price records of a stock along with the dates of the change in a list. For example, let's consider the following price records of a stock:

Price Records:
Price:  110       112       90        105
Date:   2023-06-29  2023-07-01  2023-06-25  2023-07-06

Stock price changes (sorted based on date):
Date:    2023-06-25  ->  2023-06-29  ->  2023-07-01 ->  2023-07-06
Price:      90     ->     110    ->    112    ->    105
Change:           +20           +2          -7

In this case, the biggest absolute change in the stock price was +20, which occurred between 2023-06-25 and 2023-06-29. In this case, the function should return [20, "2023-06-25", "2023-06-29"]

Two days are considered consecutive if there are no other days' data in between them in the price records based on their dates.

To assist you in testing this new function, we have provided the `TestGetBiggestChange` function.
*/



using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Diagnostics;

public class Stock
{
   public string Symbol { get; set; }
   public string Name { get; set; }

   public Stock(string symbol, string name)
   {
      Symbol = symbol;
      Name = name;
   }

   public override string ToString()
   {
      return Name;
   }
}

public class PriceRecord
{
   public Stock Stock { get; set; }
   public int Price { get; set; }
   public string Date { get; set; }

   public PriceRecord(Stock stock, int price, string date)
   {
      Stock = stock;
      Price = price;
      Date = date;
   }

   public override string ToString()
   {
      return $"Stock: {Stock} Price: {Price} date: {Date}";
   }
}

public class StockCollection
{
   public List<PriceRecord> PriceRecords { get; set; }
   public Stock Stock { get; set; }
```

```
  public StockCollection(Stock stock)
  {
    PriceRecords = new List<PriceRecord>{};
    Stock = stock;
  }

  public int GetNumPriceRecords()
  {
    return PriceRecords.Count;
  }

  public void AddPriceRecord(PriceRecord priceRecord)
  {
    if (!priceRecord.Stock.Equals(Stock))
      throw new ArgumentException("PriceRecord's Stock is not the same as the
StockCollection's");

    PriceRecords.Add(priceRecord);
  }

  public int? GetMaxPrice()
  {
    if(PriceRecords.Count>0)
    {
    return PriceRecords.Max(priceRecord => priceRecord.Price);
    }
    return null;

  }

  public int? GetMinPrice()
  {
     if(PriceRecords.Count>0)
    {
    return PriceRecords.Min(priceRecord => priceRecord.Price);
    }
    return null;
  }

  public double? GetAvgPrice()
  {
    if(PriceRecords.Count>0)
    {
```

```
        return PriceRecords.Average(priceRecord => priceRecord.Price);
        }
        return null;
    }

public void GetBiggestChange(int price, startTime, endTime)
{

}

}

public class Solution
{
    public static void Main(String[] args) {
        TestPriceRecord();
        TestStockCollection();
        TestGetBiggestChange();
    }

    public static void TestPriceRecord()
    {
        Console.WriteLine("Running TestPriceRecord");
        // Test basic PriceRecord functionality
        Stock TestStock = new Stock("AAPL", "Apple Inc.");
        PriceRecord TestPriceRecord = new PriceRecord(TestStock, 100, "2023-07-01");
        Debug.Assert(TestPriceRecord.Stock == TestStock);
        Debug.Assert(TestPriceRecord.Price == 100);
        Debug.Assert(TestPriceRecord.Date == "2023-07-01");
    }

    public static StockCollection MakeStockCollection(Stock Stock, List<Tuple<int, string>>
PriceData)
    {
        // Create a new StockCollection for test purposes.
        // Stock: The Stock object this StockCollection is for
        // PriceData: a list of tuples. Each tuple represents a price record for a single date.
        StockCollection StockCollection = new StockCollection(Stock);
        foreach (Tuple<int, string> PriceRecordData in PriceData)
        {
            PriceRecord PriceRecord = new PriceRecord(Stock, PriceRecordData.Item1,
PriceRecordData.Item2);
            StockCollection.AddPriceRecord(PriceRecord);
        }
```

```
    return StockCollection;
  }

  public static void TestStockCollection()
  {
    Console.WriteLine("Running TestStockCollection");
    // Test basic StockCollection functionality
    Stock TestStock = new Stock("AAPL", "Apple Inc.");
    StockCollection StockCollection = new StockCollection(TestStock);
    Debug.Assert(StockCollection.GetNumPriceRecords() == 0);
    Debug.Assert(StockCollection.GetMaxPrice() == null);
    Debug.Assert(StockCollection.GetMinPrice() == null);
    Debug.Assert(StockCollection.GetAvgPrice() == null);

    // Price Records:
    // Price:  110       112       90        105
    // Date:   2023-06-29  2023-07-01  2023-06-28  2023-07-06
    List<Tuple<int, string>> PriceData = new List<Tuple<int, string>>
    {
      new Tuple<int, string>(110, "2023-06-29"),
      new Tuple<int, string>(112, "2023-07-01"),
      new Tuple<int, string>(90, "2023-06-28"),
      new Tuple<int, string>(105, "2023-07-06")
    };
    TestStock = new Stock("AAPL", "Apple Inc.");
    StockCollection = MakeStockCollection(TestStock, PriceData);
    Debug.Assert(StockCollection.GetNumPriceRecords() == PriceData.Count);
    Debug.Assert(StockCollection.GetMaxPrice() == 112);
    Debug.Assert(StockCollection.GetMinPrice() == 90);
    Debug.Assert(Math.Abs((decimal)StockCollection.GetAvgPrice().GetValueOrDefault() -
104.25m) < 0.1m);
  }

  public static void TestGetBiggestChange()
  {
    Console.WriteLine("Running TestGetBiggestChange");
    // Test the get_biggest_change method
    Stock TestStock = new Stock("AAPL", "Apple Inc.");
    StockCollection StockCollection = new StockCollection(TestStock);
    Debug.Assert(StockCollection.GetBiggestChange() == (0, "", ""));

    // Price Records:
    // Price:  110       112       90        105
    // Date:   2023-06-29  2023-07-01  2023-06-25  2023-07-06
```

```
    List<Tuple<int, string>> PriceData = new List<Tuple<int, string>>
    {
      new Tuple<int, string>(110, "2023-06-29"),
      new Tuple<int, string>(112, "2023-07-01"),
      new Tuple<int, string>(90, "2023-06-25"),
      new Tuple<int, string>(105, "2023-07-06")
    };
    StockCollection = MakeStockCollection(TestStock, PriceData);
    Debug.Assert(StockCollection.GetBiggestChange() == (20, "2023-06-25", "2023-06-
29"));

    // Price Records:
    // Price:  200       210       190       180
    // Date:   2000-01-04  1999-12-30  2000-01-03  2000-01-01
    PriceData = new List<Tuple<int, string>>
    {
      new Tuple<int, string>(200, "2000-01-04"),
      new Tuple<int, string>(210, "1999-12-30"),
      new Tuple<int, string>(190, "2000-01-03"),
      new Tuple<int, string>(180, "2000-01-01")
    };
    StockCollection = MakeStockCollection(TestStock, PriceData);
    Debug.Assert(StockCollection.GetBiggestChange() == (-30, "1999-12-30", "2000-01-
01"));
  }
}
```

Question 4 – TollBooth

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;

/*
We are writing software to analyze logs for toll booths on a highway. This highway is a
divided highway with limited access; the only way on to or off of the highway is through a
toll booth.

There are three types of toll booths:
* ENTRY (E in the diagram) toll booths, where a car goes through a booth as it enters the
highway.
* EXIT (X in the diagram) toll booths, where a car goes through a booth as it exits the
highway.
* MAINROAD (M in the diagram), which have sensors that record a license plate as a car
drives through at full speed.


     Exit Booth                Entry Booth
        |                      |
        X                       E
         \                     /
---<------------<---------M---------<-----------<---------<----
                        (West-bound side)

================================================================

                        (East-bound side)
------>--------->---------M--------->--------->--------->------
         /                    \
        E                      X
        |                      |
     Entry Booth                Exit Booth


For our first task:
1-1) Read through and understand the code and comments below. Feel free to run the code
and tests.
1-2) The tests are not passing due to a bug in the code. Make the necessary changes to
LogEntry to fix the bug.

*/
```

/*
We are interested in how many people are using the highway, and so we would like to count how many complete journeys are taken in the log file.

A complete journey consists of:
* A driver entering the highway through an ENTRY toll booth.
* The driver passing through some number of MAINROAD toll booths (possibly 0).
* The driver exiting the highway through an EXIT toll booth.

For example, the following excerpt of log lines contains complete journeys for the cars with JOX304 and THX138:

.
.
.
90750.191 JOX304 250E ENTRY
91081.684 JOX304 260E MAINROAD
91082.101 THX138 110E ENTRY
91483.251 JOX304 270E MAINROAD
91873.920 THX138 120E MAINROAD
91874.493 JOX304 280E EXIT
.
.
91982.102 THX138 290E EXIT
.
You may assume that the log only contains complete journeys, and there are no missing entries.

2-1) Write a function in LogFile named CountJourneys() that returns how many
    complete journeys there are in the given LogFile.
*/

public class LogEntry
{
   /**
   * Represents an entry from a single log line. Log lines look like this in the file:
   *
   * 34400.409 SXY288 210E ENTRY
   *
   * Where:
   * * 34400.409 is the timestamp in seconds since the software was started.
   * * SXY288 is the license plate of the vehicle passing through the toll booth.
   * * 210E is the location and traffic direction of the toll booth. Here, the toll

```
   *    booth is at 210 kilometers from the start of the tollway, and the E indicates
   *    that the toll booth was on the east-bound traffic side. Tollbooths are placed
   *    every ten kilometers.
   * * ENTRY indicates which type of toll booth the vehicle went through. This is one of
   *    "ENTRY", "EXIT", or "MAINROAD".
   **/

   public double Timestamp { get; private set; }
   public string LicensePlate { get; private set; }
   public string BoothType { get; private set; }
   public int Location { get; private set; }
   public string Direction { get; private set; }

   public LogEntry(string logLine)
   {
      string[] tokens = logLine.Split(" ");
      Timestamp = Convert.ToDouble(tokens[0]);
      LicensePlate = tokens[1];
      BoothType = tokens[3];
      Location = int.Parse(tokens[2].Substring(0, tokens[2].Length - 1));
      char directionLetter = tokens[2][tokens[2].Length - 1];
      if (directionLetter == 'E')
      {
         Direction = "EAST";
      }
      else if (directionLetter == 'W')
      {
         Direction = "WEST";
      }
      else
      {
         Debug.Assert(false, "Invalid direction");
      }
   }

   public override string ToString()
   {
      return string.Format("<LogEntry timestamp: {0} license: {1} location: {2} direction: {3}
booth type: {4}>",
         Timestamp, LicensePlate, Location, Direction, BoothType);
   }
}

public class LogFile : List<LogEntry>
```

```
{
   /*
   * Represents a file containing a number of log lines, converted to LogEntry
   * objects.
   */
   public LogFile(StreamReader fileHandle)
   {
      string ?line;
      while ((line = fileHandle.ReadLine()) != null)
      {
         LogEntry logEntry = new LogEntry(line.Trim());
         Add(logEntry);
      }
   }
}

public class Solution
{
   public static void TestLogFile()
   {
      Console.WriteLine("Running TestLogFile");
      using (StreamReader fileHandle = new
StreamReader("/content/test/tollbooth_small.log"))
      {
         LogFile logFile = new LogFile(fileHandle);
         Debug.Assert(logFile.Count == 13, "Length check failed");
         foreach (LogEntry entry in logFile)
         {
            Debug.Assert(entry is LogEntry, "Type check failed");
         }
      }
   }

   public static void TestLogEntry()
   {
      Console.WriteLine("Running TestLogEntry");
      string logLine = "44776.619 KTB918 310E MAINROAD";
      LogEntry logEntry = new LogEntry(logLine);


      Debug.Assert(logEntry.Timestamp.Equals(44776.619));
      Debug.Assert(logEntry.LicensePlate.Equals("KTB918"));
      Debug.Assert(logEntry.Location.Equals(310));
      Debug.Assert(logEntry.Direction.Equals("EAST"));
```

```
        Debug.Assert(logEntry.BoothType.Equals("MAINROAD"));

        logLine = "52160.132 ABC123 400W ENTRY";
        logEntry = new LogEntry(logLine);
        Debug.Assert(logEntry.Timestamp.Equals(52160.132));
        Debug.Assert(logEntry.LicensePlate.Equals("ABC123"));
        Debug.Assert(logEntry.Location.Equals(400));
        Debug.Assert(logEntry.Direction.Equals("WEST"));
        Debug.Assert(logEntry.BoothType.Equals("ENTRY"));
    }

    public static void TestCountJourneys()
    {
        Console.WriteLine("Running TestCountJourneys");

        using (StreamReader reader = new StreamReader("/content/test/tollbooth_small.log"))
        {
            LogFile logFile = new LogFile(reader);
            Debug.Assert(logFile.CountJourneys().Equals(3));
        }

        using (StreamReader reader = new
StreamReader("/content/test/tollbooth_medium.log"))
        {
            LogFile logFile = new LogFile(reader);
            Debug.Assert(logFile.CountJourneys().Equals(63));
        }
    }

    public static void Main()
    {
        TestLogFile();
        TestLogEntry();
        TestCountJourneys();
        Console.WriteLine("All tests passed.");
    }
}
```