E-GENTING PROGRAMMING COMPETITION 2016

General instructions:

1. Answer one or more of the questions.
2. The competition is an open book test.
3. The duration of the competition is 8 hours.
4. Do not discuss matters related to the questions with other contestants.
5. To receive credit for answering a question, your answer must be a credible response to the question. A credible response is an answer that solves the problem or would be likely to solve the problem with a little additional effort.
6. Provided your answer is a credible response, you will receive credit for the products of a methodical approach. For example, data flow diagrams and state transition diagrams and tables.
7. Your total score is the sum of the credit you receive from each credible response.
8. Your programs will be assessed on the ease with which they can be read and understood.
    - Indenting must be clean and consistent.
    - Variable names should describe the contents of the variables.
    - Coupling between modules should be visible.
    - Each module should do one thing well.
9. The questions are worth the following marks:

No	Name	Marks
1.	Route Search	300
2.	Poker Analysis	300
3.	Roman Numerals	100
4.	Lift Display Controller	200

10. Unless otherwise stated, your programs may be written in any mainstream programming language under any mainstream operating system.
11. Unless otherwise stated, you may make use of all the standard library functions of your chosen language and operating system.
12. The words ‘must’, ‘must not’, ‘required’, ‘should’, ‘should not’, and ‘may’ are to be interpreted as described in RFC 2119[1].
13. You are NOT expected to answer all questions.

[1] *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, S. Bradner, March 1997.

1 ROUTE SEARCH

Faster than Light (FTL) operates a highly punctual Light Rail Transit (LRT) network in the city of Wristonia. FTL has a route control system and a route planner system. The route control system sets the routes that trains use during operation. The route planner system proposes possible routes along with estimated arrival times based on the starting time, origin station and destination station selected by users.

FTL is building new rail lines that are expected to be completed over the next few years. The new lines are designed to handle variable passenger loads and introduce asymetric routes. A train that runs along a track line may not always return in the opposite direction of the same track line. Instead trains are dynamically allocated according to forecasted commuter load at different times of the day.

The route control system is able to hande asymetric routes, but the current route planner system cannot handle asymetric routes. FTL has hired your company to update the route planner system to support asymetric routes.

Your job is to write a prototype program that finds the three earliest possible arrival times. The program must accept the following parameters:

1. starting time in seconds since midnight,
2. the station identifier of a starting station, and
3. the station identifier of a destination station.

The program must display the arrival times in seconds since the starting time.

The program may use the following database tables that are imported from the route control system:

```
create table train (
     trainId             integer not null,
     trainFirstStatId    integer not null,
     trainStartTime      integer not null
);
create unique index trainIdInd on
     train (trainId);
```

`train`
: is a table that stores the trains.

`trainId`
: is the train identifier.

`trainFirstStatId`
: is the station identifier of the first station where the train starts operation.

`trainStartTime`
: is the time the train starts operation in seconds since midnight.

```
create table route (
     routeTrainId        integer not null,
     routeSeqNo          integer not null,
     routeNextStatId     integer not null,
     routeNextWaitTime   integer not null
);
create unique index routeTrainIdSeqNoInd on
     route (routeTrainId, routeSeqNo);
```

`route`
: is a table that stores the routes.

`routeTrainId`
: is the train identifier.

`routeSeqNo`
: is the sequence number that indicates the order of the routes that the train follows. The train stops at stations in ascending sequence number order. The first sequence number is always zero. Sequence numbers are guaranteed to increment by one for each stop.

`routeNextStatId`
: is the station identifier of the next station the train stops.

`routeNextWaitTime`
: is the duration in seconds that the train stops at the next station.

```
create table track (
     trackId             integer not null,
     trackFromStatId     integer not null,
     trackToStatId       integer not null,
     trackDuration       integer not null
);
create unique index trackIdInd on
     track (trackId);
create unique index trackFromToStatInd on
     track (trackFromStatId, trackToStatId);
```

`track`
: is a table that stores the track lines. Each track line runs from one station to another in only a single direction.

`track.trackId`
: is the track identifier.

`track.trackFromStatId`
: is the station identifier of the starting station in the track line.

`track.trackToStatId`
: is the station identifier of the destination station in the track line.

`track.trackDuration`

is the duration of travel in seconds from the starting station to the destination station.

2 POKER ANALYSIS

Three-Card Poker Online (3CPO) operates a free-to-play Massively Multiplayer Online Poker Game (MMOPG). The game is played with an infinite number of card decks.

The game is played with non-monetary in-game currency and players are given free game currency daily.

Each round is played with a bet of ten currency units. In each round, the game deals three cards to each player and sorts the players according to the hand dealt to the players. The hands are prioritized in the sequence shown in Figure 1.

Hand	**Description**	**Priority**	**Example**
Three of a Kind Flush	All cards in the same suit and rank	Highest	9♣ 9♣ 9♣
Straight Flush	All cards in the same suit and in consecutive ranks	Second	A♠ K♠ Q♠
Three of a Kind	All cards in the same rank	Third	2♠ 2♥ 2♦
Straight	All cards in consecutive ranks	Fourth	4♠ 3♥ 2♠
Flush	All cards of the same suit	Fifth	10♠ 7♠ 2♠
Pair	Two cards in the same rank	Sixth	6♠ 4♠ 4♦
High Card	None of the other hands	Seventh	10♠ 9♠ 3♥

Figure 1: Hand Priority

The game always uses the hand with the highest priority when a player's cards could be interpreted as multiple hands.

The game then divides the bets into seven pools, where the bets of players with hands of the same priority go into the same pool. The game distributes the pool corresponding to hands of a given priority to all the players with hands of a higher priority. The pool corresponding to hands of the highest priority is not distributed, but is considered as house winning.

Because the game does not support decimal places in its currency, it rounds the amount paid to players. The rounding adjustments can cause the amount received by the players to not always equal the amount bet by the players with hands of a lower priority. 3CPO would like to explore a different scheme for distributing the pools to fix the rounding problem.

In the current scheme, the prize receivable by each player from a pool is rounded to the nearest integer with half rounded up. Rounding is performed before a player's prizes from each pool are added up.

In the new scheme, the prize to be received by each player from a pool is to be rounded down to the nearest integer, and the balance from the pool is to be moved to the next higher pool.

3CPO assigns you to create a program to generate reports that compare the amount given under the current scheme to the amount to be awarded under the new one. Each report must show the following information:

1. for each pool:
    a. number of players in the pool;
    b. total bets in the pool (excluding amount moved from lower pools);
    c. for the current scheme:
        i. amount paid to each player in the pool;
        ii. total amount paid to all players in the pool;
    d. for the new scheme:
        i. amount payable to each player in the pool;
        ii. total amount payable to all players in the pool;
2. grand totals for the following:
    a. players;
    b. bets;
    c. amount paid to players using the current scheme;
    d. amount payable to players using the new scheme,
3. difference between the new and the current scheme, calculated as the amount payable under the new scheme minus the amount paid under the current scheme.

Figure 2 shows a sample report.

```
                              CURRENT SCHEME        NEW SCHEME
PRIORITY  PLAYERS      POOL  / PLAYER     TOTAL  / PLAYER     TOTAL      DIFF
------- --------- --------- --------- --------- --------- --------- ---------
HIGHEST         4        40       173       692       180       720
SECOND         21       210       120     2,520       123     2,583
THIRD          55       550        98     5,390       101     5,555
FOURTH        307      3070        60    18,420        61    18,727
FIFTH         601      6010        44    26,444        44    26,444
SIXTH       1,997    19,970        24    47,928        23    45,931
SEVENTH     7,015    70,150         0         0         0         0
------- --------- --------- --------- --------- --------- --------- ---------
TOTAL      10,000   100,000         -   101,394         -    99,960    -1,434
------- --------- --------- --------- --------- --------- --------- ---------
```

Figure 2: Sample Report

The game stores its game history in files in a folder. The cards dealt in each round are stored in a single file with the identifying round number as the file name. Figure 3 shows a sample file. Each line in the file contains the cards of a single player. The cards are separated by a single space character. Each card is represented by a two-character code. The first character represents the rank as defined in Figure 4, while the second represents the suit as defined in Figure 5.

Sample Game History File	**Cards Dealt**
2s 2h 2d	2♠ 2♥ 2♦
As Ks Qs	A♠ K♠ Q♠
Xc Xc Xc	10♣ 10♣ 10♣
3s 2s 2d	3♠ 2♠ 2♦
9s 8d 7h	9♠ 8♦ 7♥
6s 5d 2d	6♠ 5♦ 2♦

Figure 3: Sample Game History File

ASCII Character	**Card Rank**
2	2
3	3
4	4
5	5
6	6
7	7
8	8

ASCII Character	**Card Rank**
9	9
X	10
J	J
Q	Q
K	K
A	A

Figure 4: Character to Rank Mapping

ASCII Character	**Card Suit**	**Suit Name**
d	♦	Diamond
c	♣	Club
h	♥	Heart
s	♠	Spade

Figure 5: Character to Card Mapping

Your program must accept the following parameters:

1. the history folder path name,
2. the first round number, and
3. the last round number.

Your program must generate a single report for each round played, from the first round to the last round. For example, if the first round number was 10 and the last round number was 12, your program should produce 3 reports. Your program must display the reports on the standard output.

3 ROMAN NUMERAL DECODER

An archaeological team has found and scanned thousands of well preserved ancient roman documents. The documents were put through an Optical Character Recognition (OCR) program and fed into various systems for analysis and processing.

Unfortunately, the system responsible for converting roman numerals to decimal numbers has encountered some problems. Many of the roman numerals in the documents do not follow the standard rule accepted by the system.

Character	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

Figure 6: Standard Roman Numerals and Values

The system only accepts the characters listed in Figure 6, but the uncommon character E that represents 250 is also found in some documents.

Characters	Value
IV	4
IX	9
XL	40
XC	90
CD	400
CM	900

Figure 7: Modern Standard Subtractive Roman Numerals and Values

The system only interprets the subtractive terms listed in Figure 7. However, non-standard subtractive terms such as those listed in Figure 8 are also found in some documents.

Without interpreting the non-standard subtractives, the system sums values of the subtractive terms and the value of the remaining characters to produce the decimal value, which is sometimes wrong.

The archeological team have come to you for help to update the system to correctly interpret the roman numerals. You have decided to write a program as a proof of concept that can interpret all the roman numerals found in the documents.

The program must read a Roman number and display the corresponding decimal value. The program is only required to support Roman numbers up to 9 characters in length.

After much consideration, you decide that the program should interpret Roman numbers in the following manner:

1. The program should sum consecutive sequences of 1 or more of the same character.
2. If a consecutive sequence is smaller than the next consecutive sequence, the program should calculate the combined value as the larger value minus the smaller value.
3. The program should combine the consective sequences from left to right, and should combine a consecutive sequence no more than once.
4. The program should calculate the Roman numeral as the sum of all uncombined consecutive sequences and all combined values.

Roman Number	**Interpretation Order**	**Values**	**Actual Value**
IIX	[(II) X]	10 – (1 + 1)	8
IIXX	[(II) (XX)]	(10 + 10) – (1 + 1)	18
IVX	[IV] X	(5 – 1) + 10	14
IIIC	[(III) C]	100 – (1 + 1 + 1)	97
IIXC	[(II) X] C	100 + (10 – (1 + 1))	108
VIC	V [IC]	5 + (100 – 1)	104
VIX	V [IX]	5 + (10 – 1)	14
VIIX	V [(II) X]	5 + (10 – (1 + 1))	13

Figure 8: Roman Numbers with Non-Standard Subtractives

4 LIFT DISPLAY CONTROLLER

Elevenator Lift Services is currently developing a new lift system for an 11-floor building. The building has three lift shafts with a lift in each shaft.

To achive maximum efficiency, the lift system is a bit unconventional. Instead of the usual “up” and “down” call buttons, each floor has 10 call buttons, numbered from 0 (for ground) to 10 but excluding the floor number where the buttons are residing. The buttons on the ground floor and the fifth floor are shown in Figure 9. These buttons allow a waiting passenger to specify his or her destination floor as the means of calling the lift.

Select destination floor		**10**
7	**8**	**9**
4	**5**	**6**
1	**2**	**3**

Select destination floor		**10**
7	**8**	**9**
3	**4**	**6**
0	**1**	**2**

Figure 9: Call Panel on Ground Floor (left) and 5th Floor (right)

Since the destination is determined when calling, the lifts do not need buttons to indicate the destination floors. Instead, each lift has 11 multi-colour LEDs, which are viewable from both inside and outside the lift, to indicate the floors where it will stop. Each LED represents a single floor. Thus, passengers are able to decide and board the lift that can get them to their destination floor.

A blue LED indicates that the lift is stopping on the floor to drop off passengers, and possibly to pick up passengers. A red LED indicates that the lift may stop on the floor to pick up passengers, depending on the scheduler’s last minute decision. An unlit LED indicates that the lift will not stop on the floor.

A single display controller controls the colour of the LEDs on all three lifts. Your job is to write the controller. You must implement the DisplayController class declared in Figure 10 to handle call requests and lift stops. The controller must use the LEDUnit interface is defined in Figure 11 to control the colour of the LEDs. You may assume an equivalent API is available in your programming language of choice, if it is not listed.

Upon initialization, the display controller must turn off all LEDs. When it becomes known that a lift no longer needs to stop on a particular floor, the display controller must switch off the LED for the floor.

C++

```
class DisplayController
{
public:
    void DisplayInit(LEDUnit light);
        // Called to initialize the controller.
    void LiftCalled(int fromFloor, int toFloor);
        // Called when a call button is pressed.
    void LiftStopped(int liftNo, int stopFloor, int direction);
        // Called when the lift stops at floor, and will be moving in the
        // specified direction (-1 = down, 0 = drop off only, 1 = up).
};
```

Java

```
public class DisplayController
{
    public void DisplayInit(LEDUnit light);
```

	`        // Called to initialize the controller.`
	`    public void LiftCalled(int fromFloor, int toFloor);`
	`        // Called when a call button is pressed.`
	`    public void LiftStopped(int liftNo, int stopFloor, int direction);`
	`        // Called when the lift stops at floor, and will be moving in the`
	`        // specified direction (-1 = down, 0 = drop off only, 1 = up).`
	`}`

C#	`public class DisplayController`
	`{`
	`    public void DisplayInit(LEDUnit light);`
	`        // Called to initialize the controller.`
	`    public void LiftCalled(int fromFloor, int toFloor);`
	`        // Called when a call button is pressed.`
	`    public void LiftStopped(int liftNo, int stopFloor, int direction);`
	`        // Called when the lift stops at floor, and will be moving in the`
	`        // specified direction (-1 = down, 0 = drop off only, 1 = up).`
	`}`

Figure 10: Lift Display Controller Declaration

C++	`public class LEDUnit`
	`{`
	`public:`
	`    virtual void LEDSwitch(int liftNo, int floor, int colour) = 0;`
	`        // Sets the colour (0=off, 1=red, 2=blue, 4=green) of the LED`
	`        // for the floor.`
	`}`

Java	`public interface LEDUnit`
	`{`
	`    void LEDSwitch(int liftNo, int floor, int colour);`
	`        // Sets the colour (0=off, 1=red, 2=blue, 4=green) of the LED`
	`        // for the floor.`
	`}`

C#	`public interface LEDUnit`
	`{`
	`    void LEDSwitch(int liftNo, int floor, int colour);`
	`        // Sets the colour (0=off, 1=red, 2=blue, 4=green) of the LED`
	`        // for the floor.`
	`}`

Figure 11: LED Unit Interface