```
Compiling submission
[1 of 3] Compiling Card             ( Card.hs, Card.o )
[2 of 3] Compiling Proj1            ( Proj1.hs, Proj1.o )
[3 of 3] Compiling Main             ( studenttest.hs, studenttest.o )
Linking studenttest ...
Beginning tests at Fri 13 Sep 2019 16:56:21 AEST
Testing submission for xuliny

Standard test case  1 (2 cards) ... 4 guesses
Standard test case  2 (2 cards) ... 4 guesses
Standard test case  3 (2 cards) ... 4 guesses
Standard test case  4 (2 cards) ... 3 guesses
Standard test case  5 (2 cards) ... 3 guesses
Standard test case  6 (2 cards) ... 4 guesses
Standard test case  7 (2 cards) ... 4 guesses
Standard test case  8 (2 cards) ... 4 guesses
Standard test case  9 (2 cards) ... 3 guesses
Standard test case 10 (2 cards) ... 4 guesses
Standard test case 11 (2 cards) ... 4 guesses
Standard test case 12 (2 cards) ... 3 guesses
Standard test case 13 (2 cards) ... 4 guesses
Standard test case 14 (2 cards) ... 3 guesses
Standard test case 15 (2 cards) ... 4 guesses
Standard test case 16 (2 cards) ... 4 guesses
Standard test case 17 (2 cards) ... 2 guesses
Standard test case 18 (2 cards) ... 4 guesses
Standard test case 19 (2 cards) ... 3 guesses
Standard test case 20 (2 cards) ... 4 guesses
Standard test case 21 (2 cards) ... 4 guesses
Standard test case 22 (2 cards) ... 3 guesses
Standard test case 23 (2 cards) ... 4 guesses
Standard test case 24 (2 cards) ... 3 guesses
Standard test case 25 (2 cards) ... 4 guesses
Standard test case 26 (2 cards) ... 4 guesses
Standard test case 27 (2 cards) ... 4 guesses
Standard test case 28 (2 cards) ... 4 guesses
Standard test case 29 (2 cards) ... 3 guesses
Standard test case 30 (2 cards) ... 4 guesses
    Hard test case  1 (3 cards) ... 5 guesses
    Hard test case  2 (3 cards) ... 4 guesses
    Hard test case  3 (3 cards) ... 3 guesses
    Hard test case  4 (3 cards) ... 5 guesses
    Hard test case  5 (3 cards) ... 4 guesses
    Hard test case  6 (4 cards) ... 6 guesses
    Hard test case  7 (4 cards) ... 4 guesses
    Hard test case  8 (4 cards) ... 5 guesses
    Hard test case  9 (4 cards) ... 5 guesses
    Hard test case 10 (4 cards) ... 4 guesses

       Standard tests attempted          :   30
       Standard tests passed             :   30
       Standard total guesses            :  109

           Hard tests attempted          :   10
           Hard tests passed             :   10
           Hard total guesses            :   45

Results Summary
```

```
            900 feedback tests     (/ 10) :   10

       Standard correctness points (/ 20) :   20
       Standard quality points     (/ 30) :   30

           Hard correctness points (/  5) :    5
           Hard quality points     (/  5) :    5

                       Total Points (/ 70) :   70
Completed tests at Fri 13 Sep 2019 16:56:23 AEST
```

```
--  File     : Proj1.hs
--  Author   : XuLin Yang 904904 <xuliny@student.unimelb.edu.au>
--  Origin   : Sat Aug 24 14:58:04 2019
--  Purpose  : An implementation of 2-player logical guessing game solution.
--
-- |This code is for providing the implementation for the two-player logical
--  guessing game. It is defined in three functions:
--      'feedback' for the respondent side
--      'initialGuess', 'nextGuess' for the guesser side.
--
--  The program is for solving the game that is the respondent have 'N'
--  selected secret cards from a deck of 52 cards without jokers for the
--  guesser to guess. The guesser first make the guess by calling
--  'initialGuess' and then the respondent response 'feedback' for the guessed
--  selection of cards and the secret selection of cards. After that the
--  guesser repeat the process by using 'nextGuess' with the respondent's
--  'feedback' until the guesser get the correct selection.
--
--  The program assume the respondent has a selection of 2-4 cards. (i.e.: N
--  range from 2 to 4 inclusively).

module Proj1 (feedback, initialGuess, nextGuess, GameState) where

import Card
import Data.List
import Debug.Trace

-- |The list of selected cards from a deck.
type Selection = [Card]

-- |The respondent's feedback, made up by (correctCards, lowerRanks,
--  correctRanks, higherRanks, correctSuits).
--  See 'feedback' function to see how each element is defined.
type Feedback = (Int,Int,Int,Int,Int)

-- |The representation of informaiton to be passed from one call of either of
--  the guess function to the other call. In order to cascading data between
--  funcitons. It records the possible selections based on the past guesses,
--  cards to be guessed, guesses have tried.
data GameState = GameState {domain::[Selection], ansNum::Int, guessesNum::Int}

-- ****************************** constants ********************************
-- |The constant for the first card in the Card Enum.
firstCard = minBound :: Card

-- |The constant for the last card in the Card Enum.
lastCard = maxBound :: Card

-- |The constant for the number of ranks in the Card Enum.
rankNum = 13

-- |The constant for no guess has been made.
zeroGuess = 0

-- |The constant for the cutoff threshold for domain space size.
domainThreshold = 1400

-- ***************************** helper function ***************************
-- |A helper function. Get the list of suit from the selection of cards.
```

```
suits :: Selection -> [Suit]
suits cards = map suit cards

-- |A helper function. Get the list of rank from the selection of cards.
ranks :: Selection -> [Rank]
ranks cards = map rank cards

-- |A helper function. Get the common elements between two lists. Each match is
--  counted once. Modify from: https://stackoverflow.com/a/27332905
--  E.g.: intersectOnce [1, 1, 2] [1, 2, 3] = [1, 2]
--        intersectOnce [1, 2, 3] [1, 1, 2] = [1, 2]
intersectOnce :: Ord a => [a] -> [a] -> [a]
intersectOnce xs ys = xs \\ (xs \\ ys)

-- |A helper function. It takes a number of cards to be chosen from the
--  remaining possible cards and remaining possible cards. It generates all
--  possible selections of `N` cards from a deck with 52 non-joker cards.
--  Note: As the order of selections is not the matter, so the successor
--  selected card will have a larger enum index. Otherwise, it is a duplicate
--  selection. In the code, drop the first possible card in the domain when
--  choosing the next card ensuring the generated selections will not have
--  duplication.
generateAllSelections :: Int -> Selection -> [Selection]
generateAllSelections 0 _ = [[]]
generateAllSelections cardNum remainingCards =
    [(x:y) | x <- remainingCards,
             y <- generateAllSelections (cardNum-1) (tail [x .. lastCard])]

-- score: expected number of remaining possible answers for the guess
-- |A helper function. It takes a list of selections (i.e.: possible guesses)
--  and returns a list of tuples consists of a score and the guess
calGuessesScore :: [Selection] -> [(Int, Selection)]
calGuessesScore guesses = [(calScore x, g) | (x, g) <- grouped]
    where
        -- divide guesses by a guess and the rest
        dividedGuesses = [(g, guesses) | g <- guesses]
        grouped = [(groupGuessAnsFeedback g as, g) | (g, as) <- dividedGuesses]

-- |A helper function. It associates a guess with a list of answers and then
--  group the associated list by the feedback of each list element.
--  In the code, `sort` before the `group` because haskell only group
--  consecutive common elements.
groupGuessAnsFeedback :: Selection -> [Selection] -> [[Feedback]]
groupGuessAnsFeedback g as = group $ sort $ [feedback a g | a <- as]

-- |A helper function. It takes grouped Feedback and return the calculated
--  by:
--  (the sum of the squares of the group sizes) / (the sum of the group sizes)
calScore :: [[Feedback]] -> Int
calScore groups = (sum (map (\x -> (length x) ^ 2) groups)) `div`
                  (sum (map length groups))

-- |A helper function. It takes 3 integers (#answer cards, #guesses guessed,
--  #guess domain) and return true iff there are 4 cards to be guessed and zero
--  guess has made or one guess has made and domain space is larger than
--  threshold. Otherwise, return false.
skipScoreCalTest :: (Int, Int, Int) -> Bool
skipScoreCalTest (ansNum, guessesNum, domainNum) = (ansNum == 4) &&
    ((guessesNum == 0) || ((guessesNum == 1) && (domainNum > domainThreshold)))
```

```
-- ***************************** major function ******************************
-- |A major function. It takes a target and a guess (in the order as described
--  in Cards.hs Line 33), each represented as a `Selection`, and returns the 5
--  feedback numbers: (correctCards, lowerRanks, correctRanks, higherRanks,
--  correctSuits), as a tuple.
feedback :: Selection -> Selection -> Feedback
feedback target guess =
        (correctCards, lowerRanks, correctRanks, higherRanks, correctSuits)
    where
        guessSuits = suits guess
        targetSuits = suits target
        guessRanks = ranks guess
        targetRanks = ranks target
        guessLowestRank = minimum guessRanks
        guessHighestRank = maximum guessRanks

        -- |The number of cards in the target are also in the guess.
        correctCards = length $ filter (\x -> elem x guess) target
        -- |The number of cards in the target having the same rank as a card in
        --  the guess.
        --  Note: X for arbitrary suit
        --      target = [QX, QX], guesses [QX]     => correctRanks = [QX]
        --      target = [QX],     guesses [QX, QX] => correctRanks = [QX]
        correctRanks = length $ intersectOnce guessRanks targetRanks
        -- |The number of cards in the target having the same Suit as a card in
        --  the guess.
        --  Note: The procedure of matched card is symmetric as the above one.
        correctSuits = length $ intersectOnce guessSuits targetSuits
        -- |The number of cards in the target having the rank lower than the
        --  lowest rank in the guess.
        lowerRanks = length $ filter (<guessLowestRank) targetRanks
        -- |The number of cards in the target having the rank higher than the
        --  highest rank in the guess.
        higherRanks = length $ filter (>guessHighestRank) targetRanks

-- |A major function. It takes the number of cards in the answer. It outputs a
--  selection of initial guess and the game state, as a tuple.
--  It assumes that the cardNum ranges from 2-4 inclusively.
initialGuess :: Int -> (Selection,GameState)
initialGuess ansNum = (firstGuess, GameState gameStateDomain ansNum zeroGuess)
    where
        -- initially game state is the whole domain
        deck = [firstCard .. lastCard]
        gameStateDomain = generateAllSelections ansNum deck

        -- In general, for an n card answer, first guess is chosen according to
        -- the guideline: ranks that are about 13/(n + 1) ranks apart are
        -- chosen and associated with different suits. 12 is ACS's index.
        rankApart = rankNum `div` (ansNum + 1)
        ans4Cards = zipWith (\s i -> Card s ([R2 .. Ace] !! i))
                        [Spade, Heart .. Club]
                        [rankApart, rankApart+rankApart .. 12]
        firstGuess = take ansNum $ ans4Cards

-- |A major function. It takes as input a pair of the previous guess and game
--  state, and the feedback to this guess, and returns a pair of the next guess
--  and new game state.
nextGuess :: (Selection,GameState) -> Feedback -> (Selection,GameState)
```

```
nextGuess (preGuess, oldGameState) preGuessFeedback = (guess, newGameState)
    where
        oldGameStateDoamin = domain oldGameState
        ansNumber = ansNum oldGameState
        guessesNumber = guessesNum oldGameState
        newGameStateDomain = filter (\x -> preGuessFeedback ==
            feedback x preGuess) $ delete preGuess oldGameStateDoamin
        newGameStateLen = length newGameStateDomain

        -- choose the middle instead of calculating score when domain space is
        -- too large
        guess = if (skipScoreCalTest (ansNumber, guessesNumber,
                    newGameStateLen))
                then
                    newGameStateDomain !! (newGameStateLen `div` 2)
                --  score: expected number of remaining possible answers for
                --  the guess. Sort the list [(score, guess)] increasingly.
                --  Thus the guess with min score is chosen.
                else
                    snd $ head $ sort $ calGuessesScore newGameStateDomain
        newGameState = GameState newGameStateDomain ansNumber (guessesNumber+1)
```