Design II
TrailBuddy Pseudocode
Group 9

PSEUDOCODE

Social & Personal stats/profile:

Friend/group matching (suggesting to friend relevant users)

```
suggestFriends(currFriends: List)
    LET suggestFriends = []
    IF currFriends.length < 1
        output "There are no friends :("
    ELSE
        for each friend in currFriends
            LET tempFriendList = friend.getFriends()
            for each f in tempFriendList
                IF currFriends.contain(f) == True
                    IF suggestFriends.Indexof(f) != -1
                        LET temp = suggestFriends.get(suggestFriends.Indexof(f))
                        temp[1]++
                    ELSE
                        suggestFriends.add([f,1])
                    ENDIF
                ENDIF
            END
        END
    ENDIF

    RETURN suggestFriends
```

Group formation

```
formGroupChat(users: List[User], groupName: String)
    LET newGroup = createGroup(groupName)
    FOR each user in users:
        inviteUserToGroup(user, newGroup)
    END
    displaySuccessMessage("Group Chat Created Successfully!")
```

Scheduling hiking meet-ups (2+ people)

```
scheduleEvent(hikeDetails: HikeDetails, participants: List[Participant])
    IF length of participants < 2
        displayErrorMessage("At least 2 participants required for a public event")
        RETURN
    ENDIF
```

```
    LET hikingEvent = createEvent(hikeDetails)
    FOR each participant in participants
        inviteParticipantToEvent(participant, hikingEvent)
    END

    FOR each participant in participants
        checkAvailability(participant, hikingEvent)
    END

    hikingEvent.eventDate = hikeDetails[0]
    hikingEvent.eventTime = hikeDetails[1]

    displaySuccessMessage("Hiking Meet-up Scheduled Successfully!")
```

Updating hike statistics after hike – “go on hike” method or sync with fitness tracker

```
updatePersonalStats(hikeDetails: HikeDetails, fitnessTrackerData: Boolean )
    IF fitnessTrackerData is True:
        LET performance = fitnessTracker.getData()
    ELSE:
        LET performance = getUserStats()  //user input (time, heart rate, etc)

        IF validateUserInput(performance):
            LET currStats = user.getUserStats()
            user.calcStats(currStats, performance, hikeDetails)//function will
calculate new stats and update
        ELSE:
            displayErrorMessage("Invalid input. Please try again.")
            RETURN
        ENDIF
    ENDIF
    hikeDetails.updateStatistics()
    displaySuccessMessage("Hike Statistics Updated Successfully!")
```

Send a Friend Request

```
sendFriendRequest(user1, user2):
    LET pending = False
    if user1 is not null and user2 is not null:
        if user1 is not friends with user2:
            pending = connect(user1, user2, pending)   //calls the connect function to
connect the users
            print("Friend request sent successfully.")
        else:
            print("Users are already friends.")
    else:
        print("Invalid users.")
```

Design II
TrailBuddy Pseudocode
Group 9

Connect friends/ Accept Request

```
connect(user1, user2, pending)
IF pending is True
    user1.friends.add(user2)    //adds the new users to the other's friend list
    user2.friends.add(user1)
    pending = false
    return true
ENDIF
```

Decline Request

```
declineRequest(user1, user2, pending)
IF pending is True
    pending = false     //ignores the friend Request since it is
being declined
ENDIF
```

Find Compatible Hikers

```
findCompatibleHikers(user: user)
LET userHistory = user.experience
LET posHikers = []
for each hike in experience
    //in this case there would be a database implementation with all the users and
    //their trail history and that data will be needed for this Function
    LET hikersList = hike.history //a list of recent hikers that completed this
specific hikers
        for each h in hikersList
            IF posHikers.contain(h)==true
                posHikers.get(h).[1]++  //increased the nuber of common hikes
            ELSE
                posHikers.add([h,1])
            ENDIF
        END
END

return posHikers
```

```
findNearbyHikers(user:user)
LET currLoc = user.location
//find users within a 15 mile radius in the database
LET userList = getUsers(15)
userList.sort() //sorting based of closest to furthest away
```

```
return userList
```

Calculate Compatibility Score between 2 users

```
calcCompatabilityScore(user1: user, user2:user)
//compare parameters age, experience, stats, location
LET score=0
IF |user1.age - user2.age| < 5
    score += |user1.age - user2.age|*2
ELSE
    score +=|user1.age - user2.age|*0.75
Endif

IF user1.experience == user2.experience
    score+= 10
ENDIF
//compare user1 stats to user2 stats
IF stats1 and stats2 similar
    score+=10
ELSE IF some similarities
    score+=5
ENDIF

IF user1.location - user2.location <15 mile //within 15 miles from each other
    score+=10
ELSE
    score+=1
ENDIF
Return score
```

INFORMATIONAL:

Providing weather for specific trails

```
weatherForTrail(trailID: String)
LET trailInfo = TrailDataAPI.getTrailInfo(trailID) //retrieves trail information
from API depending on unique trailID
      IF trailInfo is not null
            weatherForecast = WeatherAPI.getWeather(trailInfo.latitude,
      trailInfo.longitude)
            return weatherForecast
      ENDIF
END
return NO_WEATHER_FOUND
```

Design II
TrailBuddy Pseudocode
Group 9

Providing weather warnings

```
weatherWarning(weatherForecast: forecast)
LET weatherForecast be weather parameters such as temperature,
precipitation, wind speed, etc. that come from getWeatherForTrail()
    IF params.temperature > 30 degrees Celsius
        return "High temperature warning: Consider carrying extra
water and wear sunscreen."
    ENDIF
    IF params.precipitation > 50%
        return "Heavy precipitation warning: Bring rain gear and
consider postponing the hike."
    ENDIF
    IF params.windSpeed > 30 km/h
        return "High wind warning: Be cautious, especially in exposed
areas."
    ENDIF
END
return "Weather conditions are suitable for hiking. Enjoy your hike!"
```

Retrieve All Route Options / Directions

```
getMultipleRouteDirections(startLocation: String, endLocation: String)

routes = MapsAPI.retrieveAllRoutes(startLocation, endLocation)
END
return routes
```

Provide Suggested Route (depending on user preferences) (eg. easy vs difficult route)

```
getTrailReccommendations(routes: List, userPreferences: Preference)
LET suggestedRoute be the trail route that best matches the user's preferences
FOR each route in routes
      IF routeMatchesPreferences(route, userPreferences)
            suggestedRoute = route
      ENDIF
END
return suggestedRoute
```

Real-time trail updates

```
LET trailInfo = TrailDataAPI.getTrailInfo(trailID)
    IF trailInfo is not null
       LET trailUpdates = TrailDataAPI.trailUpdates(trailID)
```

```
       return trailUpdates
    ENDIF
END
return "No trail updates at this time"
```

Display Trail Statistics(distance, difficulty, elevation gain, etc.)

```
//provides the user trail statistics such as elevation gain,
difficulty, distance, etc.
trailStats(trailID):
LET trailInfo = TrailDataAPI.getTrailInfo(trailID)
trailLength = trailInfo.distance
elevationGain = trailInfo.elevationGain
difficulty = trailInfo.difficulty

    trailStatistics = {
        "Length": trailLength,
        "ElevationGain": elevationGain,
        "Difficulty": ""
    }

    IF difficulty == "Easy"
        trailStatistics.Difficulty = "Easy - Suitable for beginners"
    ELSE IF difficulty == "Moderate"
        trailStatistics.Difficulty = "Moderate - Requires decent
fitness level"
    ELSE IF difficulty == "Difficult"
        trailStatistics.Difficulty = "Challenging hike, do not
attempt without great fitness and experience."
    ELSE
        trailStatistics.Difficulty = "Information not available"
    ENDIF
END
return trailStatistics
```

BROWSING TRAILS:

Getting trails in the area

```
/** This method gives the user all the trails nearby to his/her location. The
radius here is nothing but the distance from the location of the user under
which the trails are being displayed. The distance is in miles. **/
getTrails(currentLocation: Location, radius: Distance):
```

```
  LET trailsData = TrailDataAPI.loadTrailsData()
  LET trails = []
  LET radius = 50
  for trail in trailsData:
      distance = calculateDistance(currentLocation, trail.location)
      IF distance <= radius
          nearbyTrails.append(trail)
      ENDIF
  END
  RETURN trails
```

Getting trail information

```
/** This method gives the user all the information regarding the trail selected
by the user specifically. **/
getTrailInfo(selectedTrail: Trail):
  LET trailInfo = TrailDataAPI.loadTrailsData(selectedTrail)
  IF trailInfo is not null
      RETURN trailInfo
  ELSE
      displayErrorMessage("No Info.")
  ENDIF
  RETURN null
```

Sorting trail by certain criteria

```
/** This method sorts the trails in the alphabetical order and returns the
sorted trails. **/
sortTrails(trails: List[Trail], sort: SortingCriteria):
  LET sortedTrails = sortingAlgorithm(trails, sort)
  RETURN sortedTrails
/** Thai sorting method uses quicksort to sort the trails on the basis of their
alphabetical order. **/
sortingAlgorithm(trails: List[Trail], sort: SortingCriteria):
    IF trails is empty
        RETURN []
    for trail in trails:
        trails = quickSort(trails, compareTrails(trail[i], trail[i+1])) /**
trail2 is the next trail entry in the list. **/
    RETURN trails
/** This method is for comparing two trails. **/
compareTrails(trail1: Trail, trail2: Trail):
    RETURN compare(trail1.name, trail2.name)
```

Filtering trails by certain criteria

```
filterTrails(trails: List[Trail], filter: FilterCriteria):
```

```
  LET filteredTrails = []
  for trail in trails:
      IF meetsFilterCriteria(trail, filter)
          filteredTrails.append(trail)
      ENDIF
  END
  RETURN filteredTrails
/** This method gives the specific filters in this case we have difficulty and
length of the trail that are used to get the trails accordingly. **/
meetsFilterCriteria(trail: Trail, filter: FilterCriteria):
    IF filter == Difficulty AND trail.difficulty != filter.value
        RETURN false
    ENDIF
    IF filter == Length AND trail.length > filter.value
        RETURN false
    ENDIF
    RETURN true
```

RECOMMENDATIONS:

Recommending hikes

```
/** There is a collaborative filtering machine learning formula to calculate
the similarity between users to determine the weight to put on a certain user's
rating based on their similarity to another user. Let this formula be called
cosineSimilarity with the input parameter a matrix. For further information,
refer to https://www.hindawi.com/journals/aai/2009/421425/ **/

recommendHikes(List: allHikes, matrix: userRatings)
      LET allHikes be a list of all the hikes
      LET userRatings be a matrix of all hike ratings for a user for all users
      LET userSimilarity be the matrix result of cosineSimilarity(userRatings)
      LET hikeSimilarity be the matrix result of cosineSimilarity(allHikes)
      LET recommended_hikes be a map of hike to a predicted rating
      FOR each hike h in allHikes
            INTEGRATE collaborative filtering formula to predict user rating
            for h based on distance, previous user ratings for h, user
            similarities, and hike similarities to previously rated hikes
            MAP h to result and store in recommended_hikes
      SORT recommended_hikes by decreasing rating
      RETURN recommended_hikes
```

Design II
TrailBuddy Pseudocode
Group 9

Recommending gear

```
recommendGear(Hike: hike)
      LET gearForWeather be a map of all the gear needed for all weathers
      LET gearForSeason be a map of all the gear needed for all seasons
      LET gearForDifficulty be a map of all the gear needed for all levels
      //gearForWeather, gearForSeason, and gearForDifficulty will be taken
      //from an online source/predetermined and set somewhere in a higher
      //class for consistency
      LET gearList be the list of gear needed
      ADD all gear from gearForWeather corresponding to hike.weather
      ADD all gear from gearForSeason corresponding to hike.season
      ADD all gear for gearForDifficulty corresponding to hike.difficulty
      REMOVE duplicate gear from gearList
      return gearList
```

Recommendations for specific weather

```
recommendWearher(Weather: weather)
      LET clothingForWeather be the map of clothing required for all weathers
      LET foodForWeateher be a map of the food required for all weathers
      LET miscForWeather be a map of any miscellaneous items required for all
      weather
      //clothingForWeather, foodForWeather, and miscFOrWeather will be taken
      //from an online source/predetermined and set somewhere in a higher
      //class for consistency
      LET weatherList be a list of all the needs for the specific weather
      ADD all clothing from clothingForWeather corresponding to weather
      ADD all food from foodForWeather corresponding to weather
      ADD all miscellaneous from miscForWeather corresponding to weather
      return weatherList
```

https://github.com/sprapti01/TrailBuddy