# Decidophobia

*Team 1092*
*Dian Chen, Yang Gao, Yaru Gong, Xinzhe Jiang*

## 1. Introduction

Decidophobia is an application aiming to help people to make decisions. Its different features meet different needs in people's daily lives. Basically, all our decisions in our lives can be divided into four different types: how many or what is the number should we pick, should we do it or not, what should we do, and who should do it.

Correspondingly, Decidophobia is designed to have four main features: Pick Number, Yes Or No, Turntable, Pick One. Pick Number allows people to input a domain they would like to pick from, and by shaking the phone, the user will get a random number from that domain. When people have problems deciding whether they should do something or not, the Yes Or No could help them to decide. By shaking the phone, they will get an answer of yes or no. Turntable allows people to set up different turntables by themselves depending on their requirements. For example, to decide what to eat, people can input a few choices like several restaurants they have in their mind and save them. The application will generate a turntable which has those choices on. By clicking the button, the turntable will start rolling and stop on a random answer at the end, which will be one get picked. Pick One allows people to pick a person. For example, when deciding who does the housework, each family member could put one finger on the screen, and circles with different colors will appear on the screen. After a while, the background color will change to the color of one of those circles. The person who can not see his own color under his fingers (because it has the same color as the background color) is picked to do the housework.

Decidophobia is accomplished with Android Studio. To realize all features, we utilized multiple sensors of mobile devices like touch screen sensor for multitouch, and accelerometer sensor to detect the shake of the phone. We also create a database to save users' data of their history answers.

## 2. Background and Related Works

### 2.1 Android Studio

We used Android Studio version 3.5.3 to develop our project. Android Studio, published on May 16, 2013 by Google and JetBrain, is an official integrated development environment for Google's Android operating system, which is built on IntelliJ IDEA software and designed for Android Development (Android Developer, 2020a). Meanwhile, this is the software that we utilized in class and other hands-on labs.

## 2.2 Hands-on Labs

Among all the hands-on Labs that we did in class, the following applications' tutorials that we did in previous labs are useful for our project. They are mainly in creating databases and adding intents for activities.

### *2.2.1 Activities and Intents*

Application: TwoActivities is a basic introduction for us to create a small application that contains two linking activities(Codelabs, 2020a). An `activity` in Android studio is a single screen where a user can perform a single task, such as taking a picture, sending an email, or some other similar activity. For this application, there were two screens that were loosely bound together. Each screen was represented as an activity. Normally, if a new activity starts, the previous activity will be stopped in the same devices. However, the system preserves the activity in a stack. Thus, when a new activity starts the new activity is pushed to the top of the stack and takes the user's attention. We created a main activity and inserted a button to jump to the second activity. Inside the same application folder, we created another new empty activity and inserted another button on that activity to jump back to the main activity. Inside the `.xml` file, we inserted an `onClick` event for the button to launch the second activity by adding an intent for this function call. We included the same feature for sending the second activity back to the main activity. An `intent` is used to start or activate an activity. It is an asynchronous message that developers can use in the activity file to request an action from another activity. There are two types of intent, explicit and implicit. An explicit intent is one with a known target of the intent, which the developers already know the linking activity's class name. An implicit intent is that the developers do not know the linking activity's class name, thus they created a general action to perform when they perform the required action. For this particular activity, we used explicit intent for these two activities, since we wanted to link the main and the second activity.

There is another application we made that was about implicit intent (Codelabs, 2020b). There were three buttons that sent the user to other activities. The first was to open a website, the second was to open Google map, the third was to share the text they typed in the text box. These intents do not have a specific activity class name to launch the activity, thus these are general actions when the user presses the button on the main screen.

### *2.2.2 Shared preference with database*

Application: RoomWordSample is an introduction for us to create a database in our application by utilizing Android Architecture Components to store a list of words in a Room database(Codelabs, 2020c). The following is the data structure that was used for this application. `Word` is an `entity` represented as a database table in the context of Architecture Components. `SQLite` database is used to store data, which is created and maintained by the Room persistence library. `WordDao` is a mapping of SQL queries to functions. `WordRoomDatabase` is the database layer on top of the `SQLite` database that takes care of mundane tasks that we used to handle the helper class. The `WordRoomDatabase` uses `WordDao` to issue queries to the `SQLite` database based on function called. `WordRepository` is a `class` that we create to manage multiple data sources. WordViewModel provides data to the UI. It is a communication center between the WordRepository and the UI system.  LiveData<List<Word>> is a data holder class that follows the observer pattern. This class would always hold and cache the latest updated data, and notify the observer about the data changes. Since it is a lifecycle-aware, it automatically manages stopping and resuming observing based on the activity or fragment state.

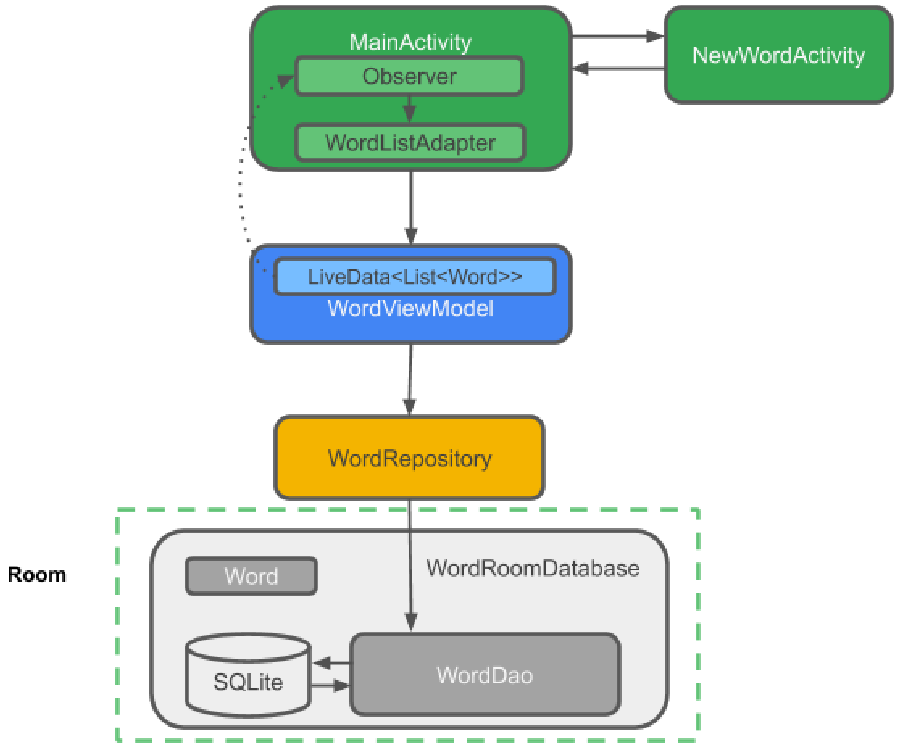


*Figure 2.1: Data Structure for RoomWordSample (Codelabs, 2020c)*

## 2.3 Sensors

Devices that are Android based have built in sensors that can measure the motion, orientation, and various environmental conditions. There are three main sensor categories that android platform supported (Android Developer, 2020c). Motion sensors measure the acceleration forces and rotational forces along three axes. Accelerometers, gravity sensors, gyroscopes, and rotational vector sensors all belong to this category. Position sensors measure the physical position of a device. Orientation sensors and magnetometers all belong to this category. Environmental sensors measure numerous environmental parameters, such as temperature, pressure, illumination and humidity. Barometers, photometers and thermometers all fall into this category.

Developers can get access to these sensors by using the sensor framework by importing the `android.hardware` package. There are four main classes inside the package. `SensorManager` is used to create an instance of the sensor service, such as registering and unregistering a sensor event listener, or accessing and listing sensors, etc. `Sensor` is used to create an instance for a specific sensor, so that the developer can acquire specific methods toward the determined sensor. `SensorEvent` is used to create a sensor event object, which includes raw sensor data, etc. `SensorEventListener` is used to create two callback methods that receive notification when sensor values or accuracy changes.

### 2.4 Existing Work

We found a similar work was done in the Apple Store for IOS users. This application is called decision maker, which has two main functionalities inside. The first functionality is choosing Yes or No. There is an animation about flipping the coin while randomly choosing Yes or No. The other functionality is a turntable that users can design their own questions and input their answers to get a "correct" answer from that.

## 3. Design and Implementation

### 3.1 Main Page

The main page (Figure 2) includes the guidance and the link to the other four features. Users can click on the image icon to check more details and use this application. For the backend, `MainActivity` includes the explicit intents for each image as a button to direct users to other activities. Users can also return to the main page by clicking the triangle back button at the bottom of the screen.

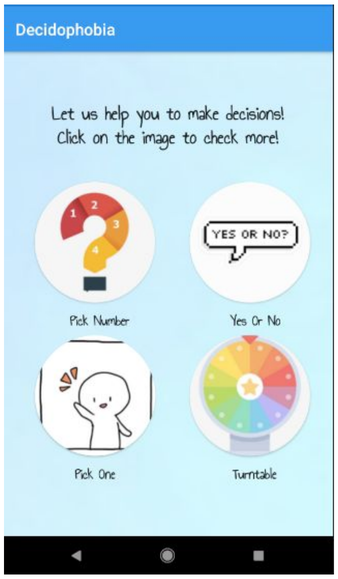
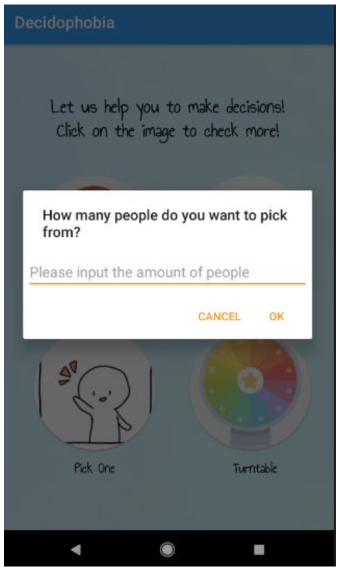

Decidophobia
Let us help you to make decisions!
Click on the image to check more!
YES OR NO?
Pick Number
Yes Or No
Pick One
Turntable

*Figure 3.1: Main Page Screenshot*

## 3.2 Pick One

The feature of `PickOne` helps the users with picking a person within the given number of people. In the beginning, the application will pump up an alert dialog and ask the number of people, and the users are supposed to input a number and click ok to the `PickOne` activity.



Decidophobia
Let us help you to make decisions!
Click on the image to check more!
How many people do you want to pick from?
Please input the amount of people
CANCEL
OK
Pick One
Turntable

*Figure 3.2 Alert dialog to ask information from users*

To make the interface more interactive with the users, we allow multiple fingers touch screen and there will be a circle with random color generated at the location of each finger. When the users touch the screen, the circle could also follow the user's finger if it moves, and if the user releases it, the corresponding circle will disappear at the same time.

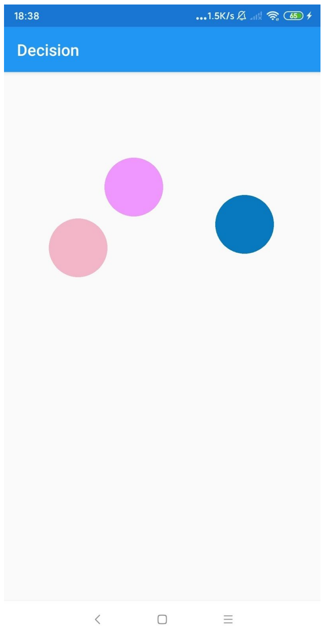


*Figure 3.3: When there are three fingers on the screen and the amount of people is four*

If the number of circles is the same as the number of the user input at the beginning, after 0.7 seconds the application will randomly pick one circle and change the background color to the corresponding circle color. After one finger or more release, it will reset and wait for enough fingers to touch the screen again.

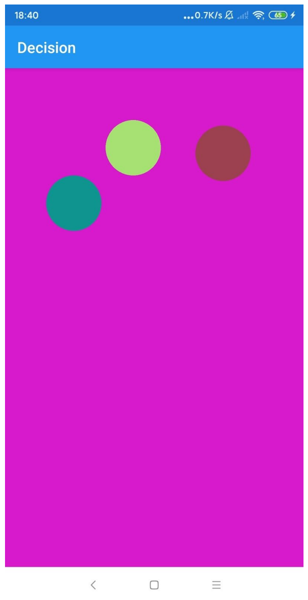


*Figure 3.4: When there are enough (four) fingers one the screen and after 0.7 second*

To realize this feature, at first, we create a dialog that asks the number of people before entering the `PickOne` activity. When the user clicks ok button, the application navigates to the `PickOne` activity and passes the number user input to the `MutiTouch` class. Then we have a class for the circle, and it has a function to draw a circle with a fixed radius and random color. We record the touch id for each touch on the screen and draw a circle at the same location as the touch event. When there are enough numbers (same number as input) of touches on the screen, the application will stop drawing circles. Instead, it will sleep for 0.7 seconds and then randomly pick a circle filling the background color with the color of that circle. If any finger releases before 0.7, the application will not change the background color until there are enough touches on the screen again and wait after 0.7 seconds.

After the background color gets changed, the user could release their fingers and repeat the process. The background color will back to white after one second if there is no new color set to the background color during that. If by anytime the users would like to change the setting of people number, they are supposed to go back to main activity and reenter `PickOne`, the dialog will pump up again.

## 3.3 Turntable

The `Turntable` activity aims to assist users making decisions on multiple choices, like finding a place to eat or a book to read. This feature implements three kinds of turntable to satisfy users' requirements, including default, self-designed, and saved turntable. After clicking the "Turntable" button, users will be led to choose those functions (Figure 6).

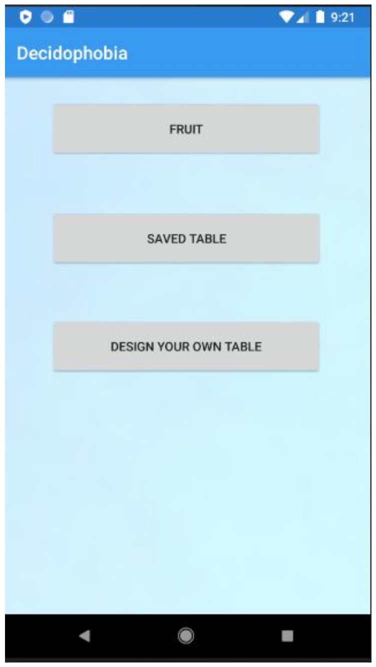


*Figure3.5: Turntable's main page*

### *3.3.1 Default Feature*

In the default feature (Figure 7), there is a pie chart which is equally divided by the number of choices with an arrow on the top of the screen pointing to one of the sectors.  Once the user pushes the "start" button, at the bottom of the screen, the turntable starts to rotate and randomly stops at some sectors.

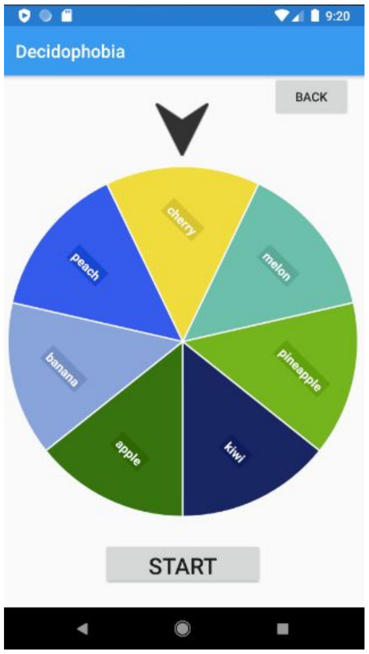


*Figure 3.6: Default turntable*

### *3.3.2 Self- designed Feature*

To make the self-designed feature, since we want to let users see what word they just add to create the turntable, we used the idea of dynamically adding views to the layout. After entering the word and clicking the "save" button (Figure 8), users stay on the same activity, but with the word shown on the screen automatically added. Considering to manage the invalid words, like empty input, we use `Toast.show()` feature to handle any invalid input, which a sentence will pop out to warn users at any time they try to enter the invalid input.

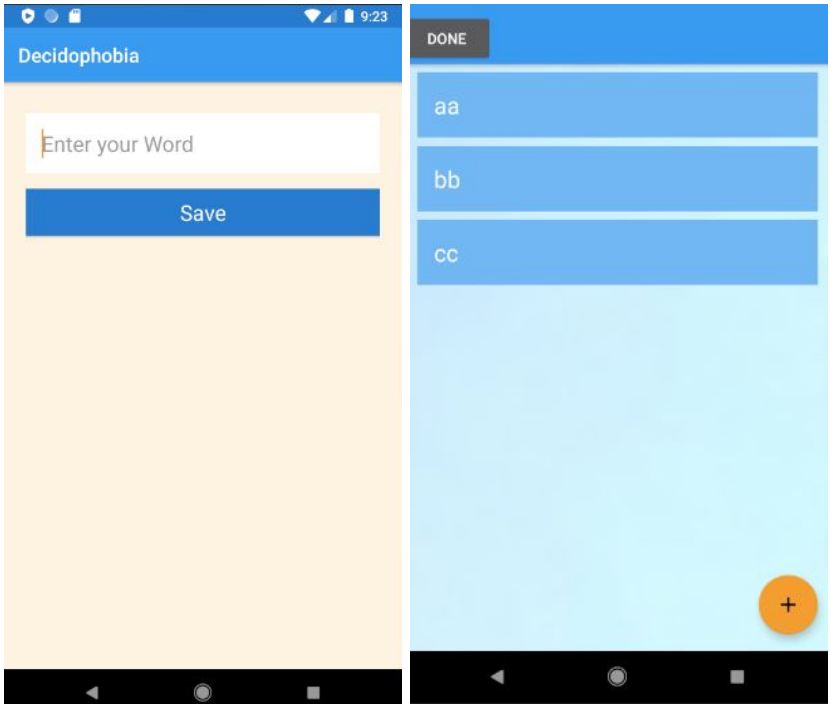


*Figure 3.7: Add Word to the word list for self-designed pie chart*

When the self-designed turntable is successfully created, the information will be collected and transferred to the "saved table" activity, which users can view and use them at any time, even restarting the application. To implement this functionality, we built a local database inside the application through Room, which is an abstraction layer over SQLite. In the table, we have an entity called "mark" to keep tracking which piece of data belongs to which turntable, an entity to record the word and a value for each word. If the "mark" equals the value of word, it means this word is used to create the current table. After users click "Done" for finishing entering words, a button with text "Table X" will be created under "saved table" activity with the information of the related table. Therefore, every time users click the "saved table" activity, all turntables are generated automatically with separate buttons.

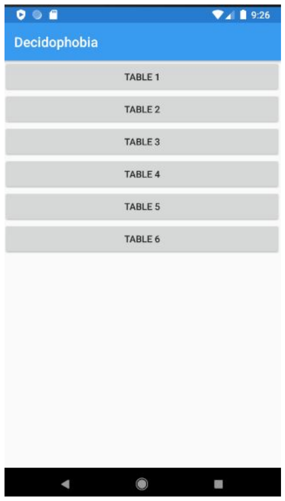


*Figure 3.8: Saved self-designed table for pie chart*

A LiveData feature is used to observe any change made on the list of words and connects with our database, which a live list can be notified in an active state if there is any change made on the list. One of advantages using `LiveData` class is to ensure the UI always matches the changes of related features, such as adding words, for instance. In addition, since the data only changes while the state is active and all data is cleaned up when its lifecycle is destroyed, there is no memory leak and crush caused by finishing an activity (Android Developer, 2020b).

This feature also requires plenty of data exchanges between different activities, which we implemented by using `putExtra()` method with Intent class. It is quite convenient to transfer data in such a way that while calling intent and launching another activity, we give a key and value we want to transfer for the method; in that intent, we can simply call method `getExtra()` with the specific key to achieve data exchange.

## 3.4 Yes or No and Pick Number

The Yes or No and Pick Number activity features to make choices for the user. The Yes or No activity will randomly generate answers between yes or no and the Pick Number activity will randomly generate numbers between the given range. One advanced feature of these two activities is that it will generate the result when you shake your phone.

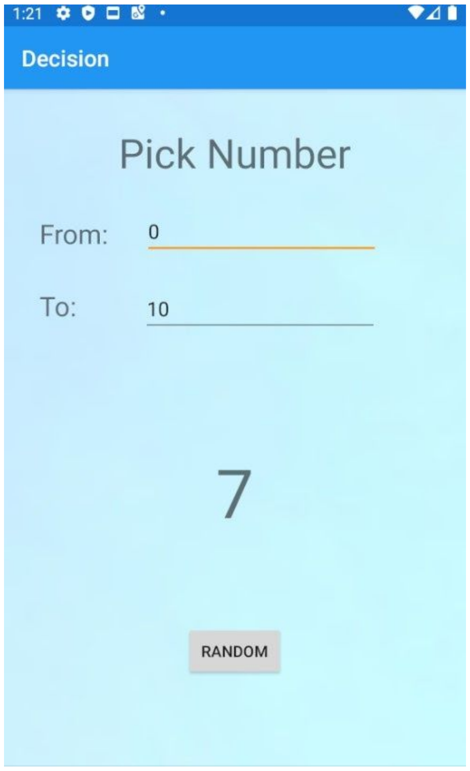


*Figure 3.9: Pick Number Feature*

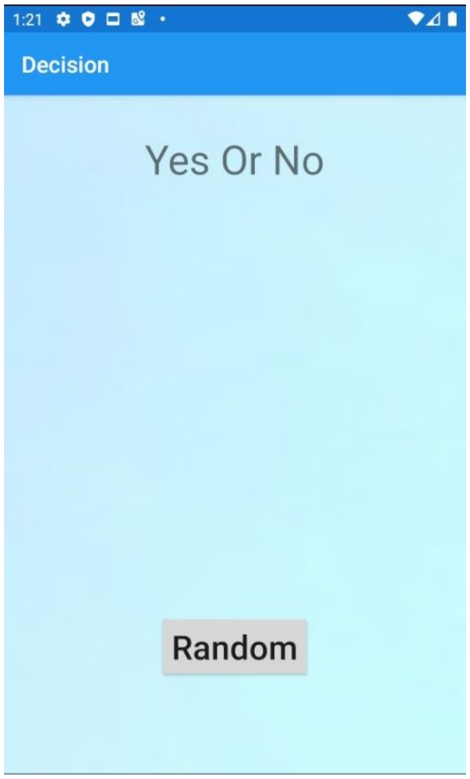


*Figure 3.10: Yes or No Feature*

### *3.4.1 Basic Feature*

It's relatively simple to achieve the basic features of these two activities. They all use the same logic that uses the built-in random function to generate the random number and display the result.

### *3.4.2 Shake Feature*

For the Shake feature in these two activities, the accelerometer sensor is used to capture the shake movement from the user. There is a class called ShakeHelper that will be used by both activities. It has onResume and onPause function written to register and unregister the accelerometer. The way ShakeHelper works is when there is a sensor change, the system will notify the ShakeHelper class. If the sensor is the accelerometer and the changed value of one of the three axises is larger than the present value, the program will broadcast an event. The activity that has the listener of that event will receive the signal that the phone has been shaken and will call the corresponding function. The sensitivity of the program can be adjusted by changing the value of the preset perimeter.

## 4. Performance Evaluation

One last thing for all developers to deal with before releasing the application is to check its performance in various ways, including application start-up, battery time while using the application, memory consumption, hardware/software variation, usage with other applications, application in background, and data to and from server(Guru99, 2020). Due to the limitation of resources, our performance evaluation only focuses on some critical aspects related to the course material. Considering the optimization target in the design of our application, we aim to optimize memory usage to maintain more utility of our application compared with other similar applications in the application store. We identify our application under the category of utilities, which combined with other retail applications has only 7% users who clarify to use most frequently compared with 39% for social media applications, 10% for gaming applications and 10% for communication/messaging applications, according to the research conducted by the Manifest(Mindsea team, 2019). As mobile application developers, we do not expect our application to take too much memory usage compared with other features in mobile devices. Besides the size of installing application itself, the memory usage while using it would increase constantly by implementing any functionality. For instance, a mobile application consumes 11% of memory while running the whole operating system only requires 14% of memory on the mobile device (Mindsea team, 2019). Such a large memory consumption may lead users to uninstall the application.

Since our application implements a local database to store users input, the best way to deal with memory usage is to find an efficient way to manage our data in our database. First, the simplest idea is only having one database through the whole application, which can reduce memory consumption while every time creating a new database. Using efficient query to search and update data is also a good strategy to both improve the cost of CPU and time consumption as well as using a transaction.

Another aspect of memory usage is to avoid memory leak, which happens, quite frequently in Java, when an object is held for a long time after it finishes the job. The consequence of memory leak is the constant increase of heap memory, which causes a bad performance of the application. Fortunately, most of the APIs and structures we use have default functions and methods to clean up themselves after calling. For example, as we described before, `LiveData` structure always self-destroys in the inactive state, which never holds useless data for too long. In addition, we use `finish()` function to stop activities running on the back stack in order to prevent multiple activities starting at the same time causing the reaction of application slow down.

The following figures show our application profile while opening and using the application, including CPU, memory, network, and energy consumption, as well as the comparison between before and after we achieve our optimization target.

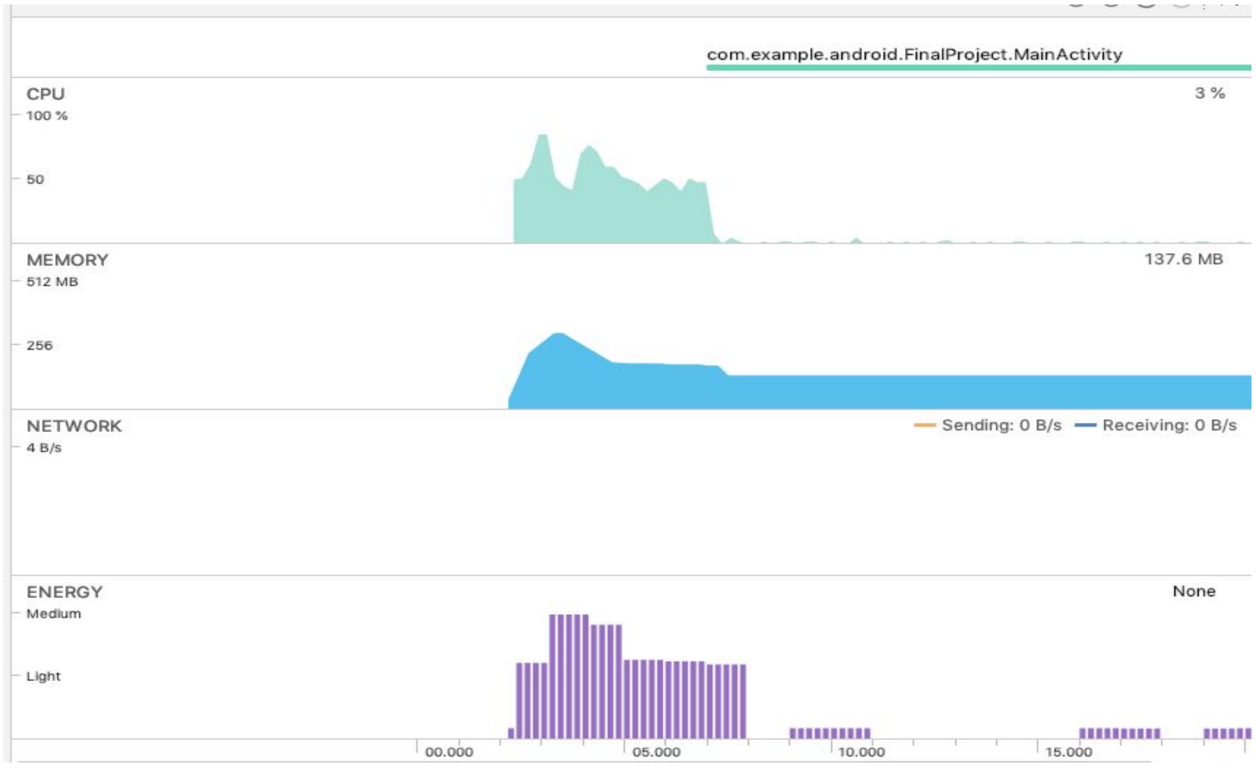


*Figure 4.1: CPU, Memory, Network, Energy consumption while using the application*

This is the profile of our application when it is opening (Figure 12). It shows that the application takes 3% of the whole CPU of mobile devices, 137.6 MB as the memory usage while opening the application, and medium energy consumption, which network data is zero since our application is totally offline. After running a while as all data movements turn to be stable, we record those data while using the application.  Since all of our optimizations work with the database, our evaluation focuses on “Turntable” feature, such as creating turntable and viewing saved tables.

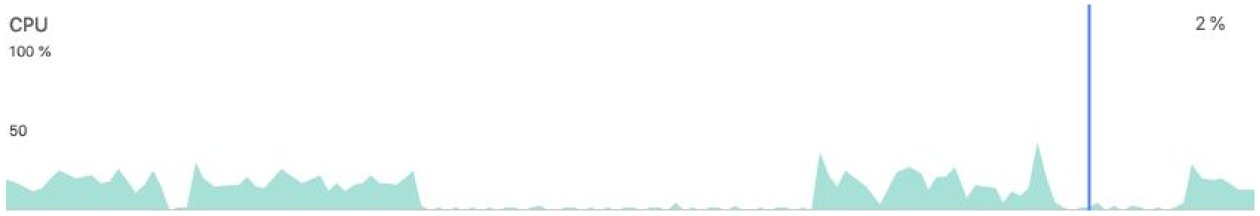


*Figure 4.2: CPU consumption for turntable*

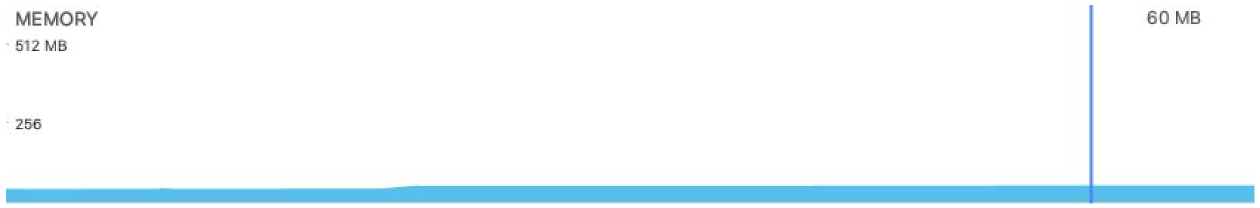


*Figure 4.3: Memory consumption for turntable*

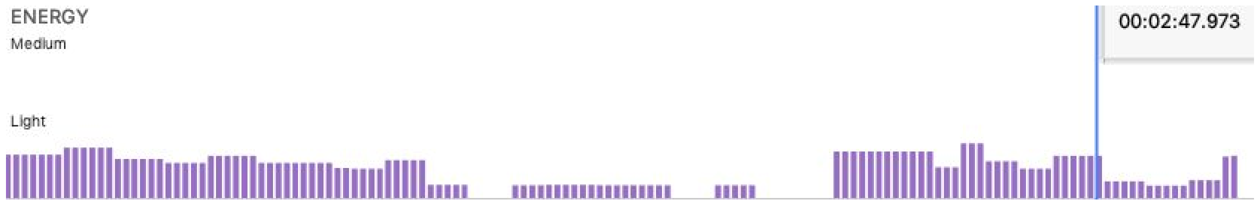


*Figure 4.3: Energy consumption for turntable*

The above figures(Figure 12, 13, 14) are the statistics about profile before we add those optimization strategies, which shows when we create a new turntable, run it and view the table again in “Saved Table” feature, it consumes 60 MB memory. The energy and CPU consumption are both in a low range.

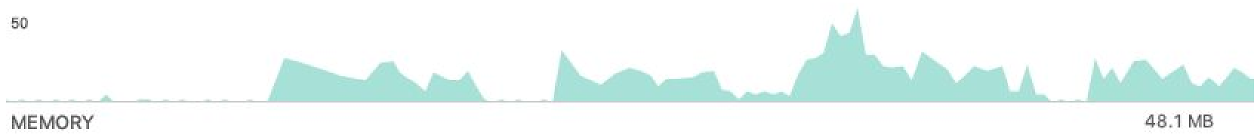
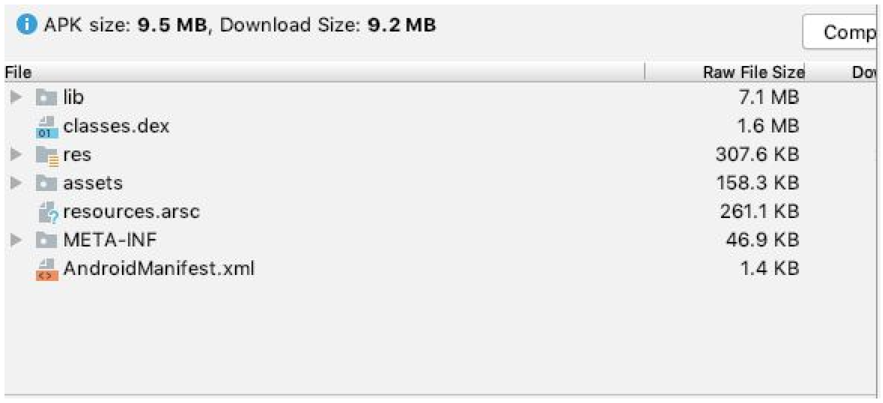


*Figure 4.4: Memory consumption for turntable after optimization*

After we implement the optimization of memory usage aspect, though the CPU and energy consumption level keep the same as before, the memory increased due to user input reduces to 48.1 MB (Figure 14).


APK size: **9.5 MB**, Download Size: **9.2 MB**

| File | Raw File Size |
|---|---|
| lib | 7.1 MB |
| classes.dex | 1.6 MB |
| res | 307.6 KB |
| assets | 158.3 KB |
| resources.arsc | 261.1 KB |
| META-INF | 46.9 KB |
| AndroidManifest.xml | 1.4 KB |

*Figure4.5: Download size of Decidophobia*

As a result, this application has a download size of 9.2 MB (Figure 15), which is reasonable compared with other similar applications we find with the same functionality. Most of the applications we check in the application store have a download size between 5 to 15 MB. For our optimization target, although the memory consumption for opening and running the application is quite large compared to its download size, our strategy regarding the database and LiveData does improve the memory usage of the application.

## 5. Conclusion

In this project, we build a complete application to serve people making decisions efficiently by implementing what we learnt in this course. We achieve our goal of building a local database to acquire, manipulate and store user data, which enables users to view their history. In addition, we create more interactions between our application and users through various sensors, such as shake and touch features. At the end, we implement several optimization strategies to improve memory usage, which provides users with a stable and convenient using experience.

## 6. Reference

[1] Android Developer. 2020a. About the platforms. Retrieved from: https://developer.android.com/about

[2] Android Developer. 2020b. LiveData overview. Retrieved from: https://developer.android.com/topic/libraries/architecture/livedata

[3] Android Developer. 2020c. Sensors overview. Retrieved from: https://developer.android.com/guide/topics/sensors/sensors_overview

[4] Codelabs. 2020a. Android fundamentals 2.1: Activities and Intents. Retrieved from: https://codelabs.developers.google.com/codelabs/android-training-create-an-activity/index.html?index=..%2F..%2Fandroid-training#0

[5] Codelabs. 2020b. Android fundamentals 2.3: Implicit intent. Retrieved from: https://codelabs.developers.google.com/codelabs/android-training-activity-with-implicit-intent/index.html?index=..%2F..android-training#0
[6] Codelabs. 2020c. Android fundamentals 10.1: Room Database. Retrieved from: https://codelabs.developers.google.com/codelabs/android-training-livedata-viewmodel/index.html?index=..%2F..%2Fandroid-training#0
[7] Guru99. 2020. Mobile App performance testing: checkList, tools. Retrieved from: https://www.guru99.com/mobile-app-performance-testing-strategy-tools.html
[8] Mindsea team. 2019. 25 mobile app usage statistics to know in 2019. Retrieved from: https://mindsea.com/app-stats/