# Quarantined Coders Big-Oh Analysis

1.

```
void MainWindow::on_pushButtonLogin_clicked()
{
    QString username = ui->lineEditUsername->text();
    QString password = ui->lineEditPassword->text();

    // Encrypte the password
    QByteArray encrypte_password (password.toStdString().c_str());
    encrypte_password.append(password);
    QString hashed_password = QCryptographicHash::hash(encrypte_password,
QCryptographicHash::Sha256).toHex();

    QSqlQuery qry;

    qry.prepare("select * from Login where username ='" +  username + "' and password ='" + password
+ "'");

    if (!qry.exec())
    {   qDebug() << "Error";   }

    if(username == "admin")
    {
        QMessageBox::information(this,"Login", "Username and Password is correct");
        changeToAdmin();
        this->ui->lineEditUsername->setText("");
        this->ui->lineEditPassword->setText("");

    }
    else if(username == "user")
    {
        QMessageBox::information(this,"Login", "Username and Password is correct");
        changetoUser();
        this->ui->lineEditUsername->setText("");
        this->ui->lineEditPassword->setText("");
    }
    else
    {
        QMessageBox::warning(this,"Login","Username and password is not correct");
        this->ui->lineEditUsername->setText("");
        this->ui->lineEditPassword->setText("");
    }
}
```

FUNCTION BIG-OH = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = O(1)

The two lines edits, `lineEditUsername` and `lineEditPassword`, read in two strings in constant time. The
string that is valid for `lineEditUsername` is “admin” or “user” whereas the string that is valid for `lineEditPassword` is “admin”. The constant checks for these two valid strings. Then proceeds to encrypt the password. If either of the line edits are incorrect or null, an “Error” will appear. This will deny them access to the program. The Big-Oh of this function is constant.

The function `on_pushButtonLogin_clicked()` has a time of
- O(1)

2.

```
void MainWindow::on_pushButtonAddNewSouvenir_clicked()
{
    if (ui->lineEditNewItem->text() == "" && ui->doubleSpinBoxNewSouvenirPrice->value() == 0.00) {

        QMessageBox::warning(this, "Invalid", "Item has no name and no value.");
    }
    else if (ui->lineEditNewItem->text() == "") {

        QMessageBox::warning(this, "Invalid", "Item has no name.");
    }
    else if (ui->doubleSpinBoxNewSouvenirPrice->value() == 0.00) {

        QMessageBox::warning(this, "Invalid", "Item has no value.");
    }
    else if (ui->labelShowItem->text() == "Souvenir Name" || ui->labelShowStadium->text() ==
"Stadium Name") {

        QMessageBox::warning(this, "Invalid", "No stadium selected.");
    }
    else {

        QMessageBox::StandardButton reply =
                QMessageBox::question(this, "Add", "Are you sure you want to add this?",
                                      QMessageBox::Yes | QMessageBox::No);

        if (reply == QMessageBox::Yes) {

            QString stadium = ui->labelShowStadium->text();
            QString newSouvenir = ui->lineEditNewItem->text();
            double newPrice = ui->doubleSpinBoxNewSouvenirPrice->value();
            m_controller->createSouvenir(stadium, newSouvenir, newPrice);

            resetSouvenirScreenLabels();
            on_comboBoxChooseStadium_activated(stadium);\
            ui->lineEditNewItem->clear();
            ui->doubleSpinBoxNewSouvenirPrice->clear();
        }
    }
}
```

This function is used to add a souvenir to the list.

Error checking occurs if there is no name and/or price input and the user receives an error message *** 1 + 1 + 1 = O(1)

When the name and price are input correctly a message box is used to ask if the information is correct. *** O(1)

The program then displays the stadium that the souvenir is being added to and updates the souvenir name and price. Once the information is input the program creates a souvenir object with the information *** O(1)

The function then sets all the ui buttons back to their original state.

The function `on_pushButtonAddNewSouvenir_clicked()` has an overall time complexity of

- O(1)

3.

```
void MainWindow::on_pushButtonCreateCustomDirectTrip_clicked()
{
    ui->stackedWidget->setCurrentWidget(ui->DirectTripScreen);

    QSqlQuery qry;
    QString query, originatedStadium, destinationStadium, resultLine;
    int totalDistance = 0;

    for (int i = 1; i < m_controller->customTripList.size(); i++) {

        originatedStadium = m_controller->customTripList[i-1];
        destinationStadium = m_controller->customTripList[i];

        qDebug() << "loop no: " + QString::number(i);

        query = "select * from [Stadium Distances] where [Originated Stadium] =
'"+originatedStadium+"' and [Destination Stadium] = '"+destinationStadium+"';";
        qry.prepare(query);

        if (!qry.exec()) {

            qDebug() << "error in on_pushButtonCreateCustomDirectTrip_clicked";
        }
        else {

            if (qry.first()) {

                totalDistance += qry.value(2).toInt();
                qDebug() << qry.value(0).toString() + " to " + qry.value(1).toString();
                resultLine = qry.value(0).toString() + " to " + qry.value(1).toString() + " is " +
qry.value(2).toString();
                ui->textBrowserDisplayDirectTrip->append("jeff" + QString::number(i));
                qry.clear();
            }
        }
     }

    ui->labelShowTotalDistanceFromDirectTrip->setText(QString::number(totalDistance));
}
```

This function creates a custom trip using input from the user. After the user inputs all the information needed the function creates variables for the data. A for loop is used with the parameters running in O(1)

The `originatedStadium` is a vector that stores the 1st stadium input then `destinationStadium` is updated with the next stadium input. O(1)

The function then uses the for loop to calculate the total distance of the custom trip storing it in the variable `totalDistance` . O(1)

The function `on_pushButtonCreateCustomDirectTrip_clicked()` has an overall time complexity of

- O(1)

4.

```
int Graph::getTotalDistance()
{
    for (int i = 0; i < adjList.size(); ++i)
    {
        for(int j = 0; j < adjList[i].adjacent.size(); j++)
        {
            if(adjList[i].adjacent[j].type == DISCOVERY)
            {
                totalDistance += getWeight(adjList[i].name, adjList[i].adjacent[j].destination);
            }
        }
    }

    return totalDistance;
}
```

This function gets the total distance using nested for loops that both us a constant to exit the loop O(1)
Within the inner for loop the total distance is added to the weight of the graph. To do this it calls on the get weight function which also uses nested for loops that run in constant time. O(1)
The function has an overall time complexity of

- O(1)

5.

```
void recursiveDijkstra(QString vertex,int position, int length)
{
    adjList[findVertexIndex(vertex)].visited = true;

    if(position != length)
    {
        Dijkstra(vertex);
        location = findSmallest();
        shortestDistance += adjList[location].distance;

        order2.append(adjList[location].name);
        recursiveDijkstra(adjList[location].name, position + 1, length);
    }
}
recursiveDijkstra(vertex, n)
```

This function uses recursion for the Dijkstra. Using the adjList to find whether the vertex has been found. If statement is implemented to exit the recursive function. O(1)
`Dijkstra` is called with the parameter vertex which is data type string. O(log n)
Location and shortestDistance variables are updated with complexity of O(1)
This function has an overall time complexity of
- O(log n)