Welcome to Foobar! 🚀 Are you ready to tackle a thrilling programming exercise for our favorite social network, Foobar? 🤩

**Instructions:**

1. Read the exercise to the end.
2. Perform a sprint planning session with your team in which you will:
    a. Identify the tasks that need to be performed.
    b. Create a dedicated Jira project and create the aforementioned tasks, organizing them into relevant Epics.
    c. Create a sprint and add all the tasks to it.
3. Start the sprint and start working on the tasks, maintaining their status. Namely, when you start a task, move it to "in progress" and when you finish a task, move it to "done".
4. Continue until the sprint is completed and submit the exercise.

**Steps:**

1. The first step is to create a project that supports writing tests. I will give you **detailed** instructions to achieve that. The instructions will be for a Linux environment. If you choose to work in another environment, you will need to use the Internet to understand how to set up an equivalent project. You will find in the Moodle a short demo video of me installing the project as explained here, if you are interested.
2. The second step will be to implement a Bloom filter in C++ following TDD.
3. The third and final step will be to deploy your code to DockerHub.

## **Step 1: project setup.**

1. Create a new **private** GitHub repository with a README file and .gitignore.
2. **Clone** the repository locally (to your computer).
3. Create the following file:
   a. CMakeLists.txt
4. Create the following directories:
   a. src (this directory will contain our 'real' code ⇒ the code we want to test)
   b. tests (this directory will contain the code that tests our 'real' code)

We will now create a few dummy files, to make sure everything works and to check that we can perform unit-testing using Google Test.

5. Create a calc.cpp file in the src directory. Inside calc.cpp, create a function that takes two integers and returns their sum:

   ```
   int sum(int a, int b){
      return a+b;
   }
   ```

   **This is the code we would like to test.**

6. Create a calc_test.cpp file in the tests directory. Inside calc_test.cpp, add a test that will perform a sanity check for the sum function:

   ```
   #include <gtest/gtest.h>
   #include "../src/calc.cpp" // here we include the code to be tested

   TEST(SumTest, BasicTest) {
      EXPECT_EQ(sum(1,2), 3);
   }
   ```

7. Create a test_main.cpp file in the tests directory. Inside test_main.cpp, add the following code that will allow to run all the tests we define:

   ```
   #include <gtest/gtest.h>
   ```

```
int main(int argc, char** argv) {
   ::testing::InitGoogleTest(&argc, argv);
   return RUN_ALL_TESTS();
}
```

8. Add the following lines to your CMakeLists.txt file:

```
cmake_minimum_required(VERSION 3.14)
project("MyCalc") # Replace “MyCalc” with your repo name

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include(FetchContent)
FetchContent_Declare(
 googletest
 URL
https://github.com/google/googletest/archive/03597a01ee50ed
33e9dfd640b249b4be3799d395.zip
)

enable_testing()

set(SOURCE_FILES
   src/calc.cpp # this is where the code we want to test
)

set(TEST_FILES
   tests/test_main.cpp # this is the main for tests
   tests/calc_test.cpp # this file contain the tests
)

add_executable(CalcTests ${TEST_FILES})
```

```
# Link Google Test to the test executable
target_link_libraries(CalcTests PRIVATE gtest gtest_main)

# Include the source directory for tests
target_include_directories(CalcTests PRIVATE src)

include(GoogleTest)
add_test(NAME CalcTests COMMAND CalcTests)
```

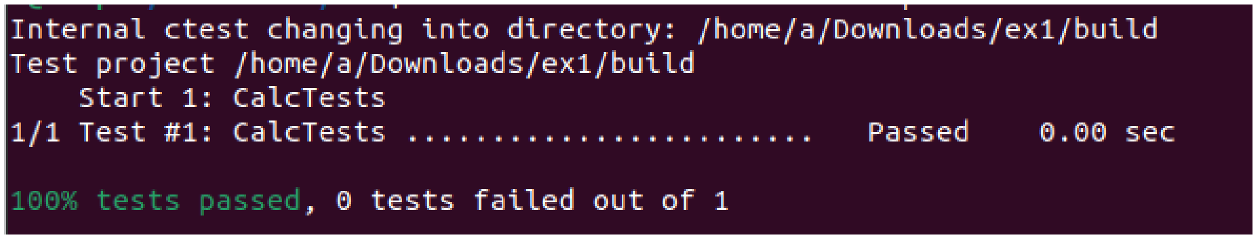
9. Install the required libraries in Linux in order for the project to work. Open a terminal and execute:
   ```
   sudo apt-get update
   sudo apt-get install libgtest-dev cmake
   ```

10. Make sure that your terminal is in the repository's directory. For example, my repository is in my desktop in a folder called ex1, so, I navigated to it using:
    cd ~/Desktop/ex1/

11. Build the project:
    ```
    cmake -B build -S .
    cmake --build build
    ```
    These commands will add a build folder to your repo. Put that folder in the .gitignore file

12. Now, we can run the tests:
    ```
    ctest --test-dir build --output-on-failure
    ```

13. This is what you suppose to see:



```
Internal ctest changing into directory: /home/a/Downloads/ex1/build
Test project /home/a/Downloads/ex1/build
    Start 1: CalcTests
1/1 Test #1: CalcTests ........................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1
```

Change the sum function in calc.cpp from + to -. If you re-run the test command from line 12, you will see the same result - **even though the function now is supposed to fail the test**.

You need to re-run step 11 to compile the changes you do in your code, and then step 12 will run the tests.

Try it.

14. You now have a working project which allows you to write code and tests, and also run them. You no longer need calc.cpp and calc_test.cpp, you can delete them. In the next step of the exercise, you are asked to implement your code in the src/ folder and tests/ folder. Note, that you might need to edit the CMakeLists.txt to reflect the code you add.

**End of step 1.**

**<u>Step 2:</u>**

## 🔍 The Bloom Filter Quest 🔍

Objective: In this challenge, you will be diving into the fascinating world of Bloom filters and helping Foobar implement a URL filtering system to protect our beloved users from blacklisted websites.

### 🌼 What's a Bloom Filter, you ask? 🌼

Imagine it's like a magical sieve for URLs! 🪄 A Bloom filter is a probabilistic data structure that's super speedy and memory-efficient. It helps us determine if an item (in our case, a URL) is a member of a set or not, without storing the actual items themselves.

### 🌼 How does it work, you wonder? 🌼

1. We have an array of bits. Initially, all bits equal zero:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

2. To blacklist a URL, we add the URL to the Bloom filter by hashing it multiple times and mark the corresponding bits in the array. For example, let's say we want to blacklist the URL www.example.com0. To add it to our Bloom filter we hash it using two hash functions h1 and h2:
   - h1("www.example.com0") = 7101593297690918553
   - h2("www.example.com0") = 2725313165333533642

Since our bit array is only 8 bits long, we do a modulo operation on the results:

- 7101593297690918553 % 8 = 1
- 2725313165333533642 % 8 = 2

Take the results and mark the bits on that indexes as 1:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

3. To check if a URL is blacklisted:
   3.1 Hash it the same way as described in step 2.
   3.2 Check if all the corresponding bits equal 1.
   3.3 If they are, we **might** have a match; if not, it's definitely not in our blacklist.

**Examples:**

In this exercise we will use two different hash functions: std::hash (h1) and (h2).

To check if www.example.com0 was blacklisted, do:

- h1(“www.example.com0”) = 7101593297690918553 % 8 = 1
- h2(“www.example.com0”) = 2725313165333533642 % 8 = 2

Since both of these indexes in the bit array equal 1, this is a blacklisted URL (not surprising… we just added it!)

Lets check if www.example.com1 was blacklisted:

- h1(“www.example.com1”) = 392235495511934953 % 8 = 1
- h2(“www.example.com1”) = 2725313165333533643 % 8 = 3

The array in index 1 is indeed equal 1 but on index 3 there is a 0, and therefore, this is NOT a blacklisted URL.

But what about the following:

- h1(“www.example.com7”) = 3258564634558621225 % 8 = 1
- h2(“www.example.com7”) = 2725313165333533649 % 8 = 1

Weird!!!! we never added it to the list! So, why is it blacklisted?

## 💡 The catch? 💡

Bloom filters can have false positives. Namely, there might be urls which the Bloom filter will classify them as blacklisted, but they in fact are not blacklisted. But we'll never have a false negative, so we stay on the safe side!

To address false positives we will:

- Use longer bit arrays.
- Double check against the real list of blacklisted URLs for any result which is positive - to ensure it is not a false positive.

## 🚀 Your Mission: Test-Driven Development (TDD) for Your Bloom Filter in C++ 🚀

You need to implement a C++ program that:

- Loops forever.
- In every iteration, it takes an entire line as input.
- If the line is of the form: 1 [URL]
  You need to add [URL] to the list of blacklisted urls
- If the line is of the form: 2 [URL]
  You need to output true or false whether it is blacklisted according to the bloom filter. This means there could be false-positive. Therefore, if it is true, also check if it's a false positive.

The first line of input will be the bloom filter array size and which functions to use.
Namely, we can use many different hash functions and not specific ones.
For example, if the first line of input is 100 1, this means we have an array of size 100 bits and we only use 1 hash function, which is the built-in std::hash.

For example, If the first line of input is 128 2, this means we have an array of size 128 bits and we only use 1 hash function, but this hash function is performing std::hash twice.
For example, If the first line of input is 256 2 1, this means we use two hash functions, where the one is std::hash and the second one performs std::hash twice.

If you get an input line in a wrong format, ignore it and read the next line.

Your output must be IDENTICAL to the output in the following examples.
**No additional text/instructions.**
**No spaces, new lines, upper casing, wrong spelling etc.**

**Input/output example 1:**

```
a
8 1 2
2 www.example.com0
false
x
1 www.example.com0
2 www.example.com0
true true
2 www.example.com1
false
2 www.example.com11
true false
```

**Input/output example 2:**

```
8 1
1 www.example.com0
2 www.example.com0
```

```
true true
2 www.example.com1
true false
```

__**Input/output example 3:**__

```
8 2
1 www.example.com0
2 www.example.com0
true true
2 www.example.com4
true false
```

**Remember:** in TDD, you write tests to specify how your code should behave __**before**__ you actually implement the solution. Use the Red-Green-Refactor methodology and add more tests and code as needed until all your tests pass. TDD is an iterative process, so you'll often find yourself going back and forth to refine your code.

By following TDD, you'll ensure that your Bloom Filter behaves as expected and maintains its integrity while evolving.

Think how to improve your implementation during the refactor steps. For example, how to make your implementation loosely-coupled with the number of hash functions and their specific implementations. In the future, we might want to change the number of hash functions used and the specific implementation of these hash functions. In TDD - this process is built-in. You first implement a naive implementation which directly calls one hash function specifically, and then - you refactor it to be flexible to support any number of functions without caring which functions. You need to document that you did this process (in your commits).

🌟 **Ready to Bloom?** 🌟

Get creative, have fun, and build a rock-solid Bloom filter to protect Foobar users from the dark corners of the internet! 🌐

**End of step 2.**

**Step 3:**

We now have a successful Bloom filter based URL filtering application. We would like to be able to run it on any computer and have a CI/CD process to streamline the future development and delivery of our product.

The problem is that there are several things that our app relies on.
Remember all the commands you had to execute in steps 1& 2?
How can we ensure that the computer we want to execute our code on has C++ compiler? What about other configurations?

Create Github workflows that:

- For every pull request into the main branch will run all the tests from step 2.
- For every release version of the code will automatically push a dockerized version of your code to a private repository on DockerHub.

**End of step 3.**

## הנחיות הגשה:

אין להשתמש בספריות מוכנות או בקטעי קוד מוכנים אלא לממש בעצמכם את הקוד, מבנה הנתונים והפונקציות שלו. יש לממש במבנה קוד תקין כלומר הצהרות בקובץ h בנפרד מהמימוש בקבצי cpp. יש להקפיד לתעד את הקוד לכל אורכו. במידה ושם המחלקה/משתנה/פונק׳ מורכב ממספר מילים, האות הראשונה של כל מילה היא אות גדולה. שמות של מחלקות חייבים להתחיל באות גדולה. שמות של משתנים ופונק׳ מתחילים באות קטנה.

חובה לעבוד בגיט לאורך כל התרגיל. כל סטודנט עובד מהמחשב שלו ומהמשתמש הנפרד שלו. לא ניתן לעבוד ביחד מאותו מחשב או מאותו משתמש או ללא גיט. חובה לעבוד בשיטת feature branches בלבד. חובה לתכנן ולחלק את המשימות בJIRA, ולתחזק אותן בהתאם להתקדמות בפרויקט. אלה דרישות מהותיות ומשקלם בתרגיל משמעותי.
חובה להגיש את התרגיל בזוגות או בשלשות. הגשה ביחידים (מכל סיבה שהיא), תגרור תקרה של ציון מקסימלי של 75 בכל תרגיל שיוגש ביחידים. לא ניתן יהיה לקבל ציון מעל 75 בתרגיל שהוגש ביחידים.

את התרגיל יש להגיש למודל לתיבת ההגשה על ידי הגשת קובץ טקסט בשם details.txt עם שמות ות.ז. של המגישים וקישור לגיט. שימו לב, חובה על הקובץ להיות בפורמט הבא:

```
Israel Israeli 123456789
Israela Israeli 012345678
Israeli Israeli 012345679
LINK TO GITHUB HERE
```

בלי רווחים נוספים, בלי שורות נוספות, ובשפה האנגלית בלבד. אי הגשה של קובץ ה details.txt הנ״ל או הגשתו באופן שונה ממה שהוגדר, **תגרור הורדה של 20 נקודות בציון התרגיל.**

אנא דאגו לכך שהגיט שלכם יהיה במצב private לכל אורך הסמסטר. יש להוסיף את מייל הקורס עם הרשאות צפייה בGithub ובjira.

שאלות לגבי התרגיל יש לשאול בפורום בלבד. פניות בנושאים פרטיים יש לשלוח למייל הקורס.
העתקות יבדקו ע״י מערכת אוטומטית, ויטופלו ישירות על ידי המחלקה.

על הגיט להכיל קובץ README שמסביר איך לקמפל ואיך להריץ את הקוד וכן מסביר את תהליך העבודה שלכם בקצרה (jira, tdd, docker) ואיזה refactorים עשיתם.

**בהצלחה!**