Lab1

Yiwen Xu

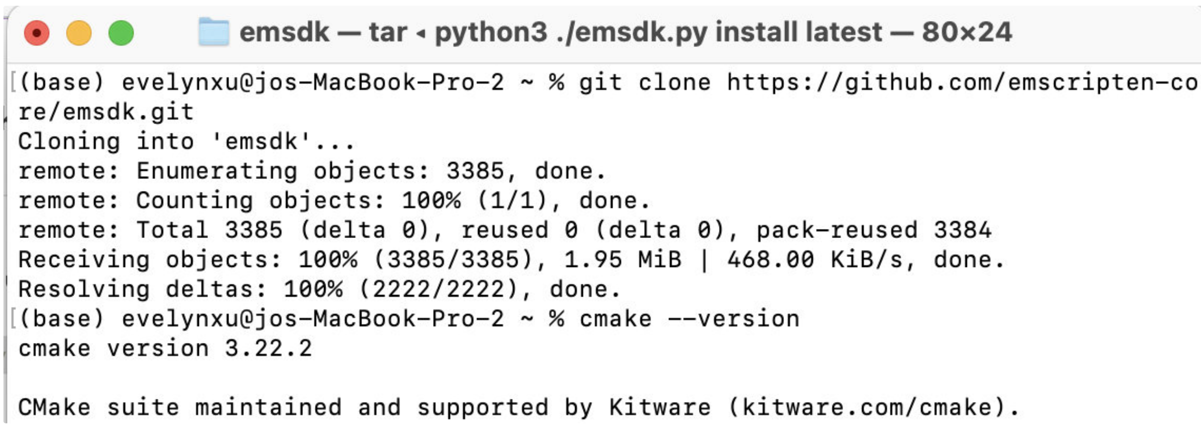
48377645

Setup the Emscripten

I completed the installation by following the instructions in the official documentation.

Download and install using emsdk

```
# Get the emsdk repo
git clone https://github.com/emscripten-core/emsdk.git
```




```
# Fetch the latest version of the emsdk (not needed the first time)
git pull


# Download and install the latest SDK tools.
# Will installed nodejs, python, etc.
./emsdk install latest

# Make the "latest" SDK "active" for the current user. (writes .emscripten file)
./emsdk activate latest

# Activate PATH and other environment variables in the current terminal
source ./emsdk_env.sh
```

- Install

```
(base) evelynxu@jos-MacBook-Pro-2 emsdk % ./emsdk install latest
Resolving SDK alias 'latest' to '3.1.20'
Resolving SDK version '3.1.20' to 'sdk-releases-upstream-d92c8639f406582d70a5dde27855f74ecf602f45-64bit'
Installing SDK 'sdk-releases-upstream-d92c8639f406582d70a5dde27855f74ecf602f45-64bit'..
Installing tool 'node-14.18.2-64bit'..
Downloading: /Users/evelynxu/emsdk/zips/node-v14.18.2-darwin-x64.tar.gz from https://storage.googleapis.com/web
assembly/emscripten-releases-builds/deps/node-v14.18.2-darwin-x64.tar.gz, 32076686 Bytes
Unpacking '/Users/evelynxu/emsdk/zips/node-v14.18.2-darwin-x64.tar.gz' to '/Users/evelynxu/emsdk/node/14.18.2_6
4bit'
Done installing tool 'node-14.18.2-64bit'.
Installing tool 'python-3.9.2-64bit'..
Downloading: /Users/evelynxu/emsdk/zips/python-3.9.2-3-macos-x86_64.tar.gz from https://storage.googleapis.com/
webassembly/emscripten-releases-builds/deps/python-3.9.2-3-macos-x86_64.tar.gz, 31899321 Bytes
Unpacking '/Users/evelynxu/emsdk/zips/python-3.9.2-3-macos-x86_64.tar.gz' to '/Users/evelynxu/emsdk/python/3.9.
2_64bit'
Done installing tool 'python-3.9.2-64bit'.
Installing tool 'releases-upstream-d92c8639f406582d70a5dde27855f74ecf602f45-64bit'..
Downloading: /Users/evelynxu/emsdk/zips/d92c8639f406582d70a5dde27855f74ecf602f45-wasm-binaries.tbz2 from https:
//storage.googleapis.com/webassembly/emscripten-releases-builds/mac/d92c8639f406582d70a5dde27855f74ecf602f45/wa
sm-binaries.tbz2, 349627729 Bytes
Unpacking '/Users/evelynxu/emsdk/zips/d92c8639f406582d70a5dde27855f74ecf602f45-wasm-binaries.tbz2' to '/Users/e
velynxu/emsdk/upstream'
Done installing tool 'releases-upstream-d92c8639f406582d70a5dde27855f74ecf602f45-64bit'.
Done installing SDK 'sdk-releases-upstream-d92c8639f406582d70a5dde27855f74ecf602f45-64bit'.
```

- Activate SDK

```
(base) evelynxu@jos-MacBook-Pro-2 emsdk % ./emsdk activate latest
Resolving SDK alias 'latest' to '3.1.20'
Resolving SDK version '3.1.20' to 'sdk-releases-upstream-d92c8639f406582d70a5dde278
55f74ecf602f45-64bit'
Setting the following tools as active:
   node-14.18.2-64bit
   python-3.9.2-64bit
   releases-upstream-d92c8639f406582d70a5dde27855f74ecf602f45-64bit

Next steps:
- To conveniently access emsdk tools from the command line,
  consider adding the following directories to your PATH:
    /Users/evelynxu/emsdk
    /Users/evelynxu/emsdk/node/14.18.2_64bit/bin
    /Users/evelynxu/emsdk/upstream/emscripten
- This can be done for the current shell by running:
    source "/Users/evelynxu/emsdk/emsdk_env.sh"
- Configure emsdk in your shell startup scripts by running:
    echo 'source "/Users/evelynxu/emsdk/emsdk_env.sh"' >> $HOME/.zprofile
```

- Activate PATH

```
(base) evelynxu@jos-MacBook-Pro-2 emsdk % source ./emsdk_env.sh
Setting up EMSDK environment (suppress these messages with EMSDK_QUIET=1)
Adding directories to PATH:
PATH += /Users/evelynxu/emsdk
PATH += /Users/evelynxu/emsdk/upstream/emscripten
PATH += /Users/evelynxu/emsdk/node/14.18.2_64bit/bin

Setting environment variables:
PATH = /Users/evelynxu/emsdk:/Users/evelynxu/emsdk/upstream/emscripten:/Users/evely
nxu/emsdk/node/14.18.2_64bit/bin:/Users/evelynxu/opt/anaconda3/bin:/Users/evelynxu/
opt/anaconda3/condabin:/Library/Frameworks/Python.framework/Versions/3.9/bin:/usr/l
ocal/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/Library/Apple/usr/bin
EMSDK = /Users/evelynxu/emsdk
EM_CONFIG = /Users/evelynxu/emsdk/.emscripten
EMSDK_NODE = /Users/evelynxu/emsdk/node/14.18.2_64bit/bin/node
EMSDK_PYTHON = /Users/evelynxu/emsdk/python/3.9.2_64bit/bin/python3
SSL_CERT_FILE = /Users/evelynxu/emsdk/python/3.9.2_64bit/lib/python3.9/site-package
s/certifi/cacert.pem
```

Verifying the installation

```
emcc -- version
```

- Verify

```
(base) evelynxu@jos-MacBook-Pro-2 emsdk % emcc --version
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 3.1.20 (5d87
8c99921ec247d34fb26a20b5a13d60d69e93)
Copyright (C) 2014 the Emscripten authors (see AUTHORS.txt)
This is free and open source software under the MIT license.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURP
OSE.
```

Running Emscripten

We can now compile C/C++ file to JavaScript.

Note: Our newly opened terminal window is not capable of executing `emcc` and needs to be activated each time with the following command.

```
cd emsdk
./emsdk activate latest
source ./emsdk_env.sh
```

First, we create a C++ file: `hello_world.cpp`.

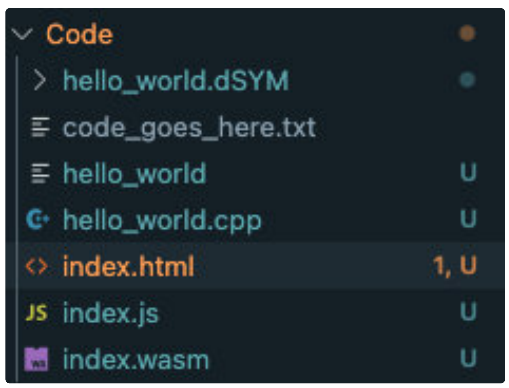
```
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]){
    cout << "hello world\n";
    return 0;
}
```

To build the JavaScript version of this code. We open a new terminal, active emcc. Go from this terminal window to the folder where you saved this C++ file. Then execute the following command in the terminal.

```
emcc hello_world.cpp -o index.html
```

After that, we get three files:

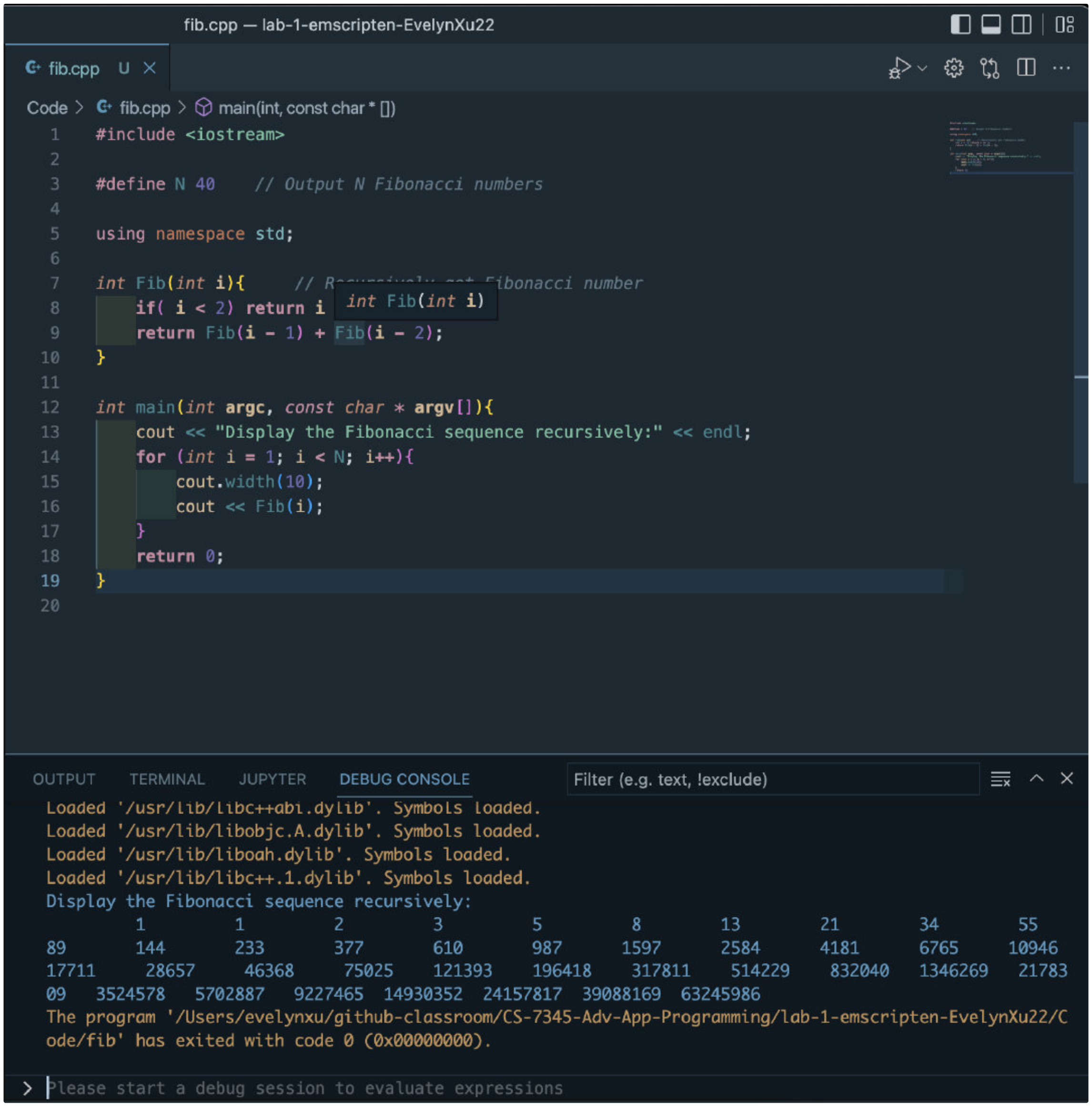
- index.html
- index.js
- index.wasm



Open the HTML file through Live Sever. We can see hello world shows correctly on this page.

Identify code base

I wrote a Fibonacci output function based on C++ language as a code base.

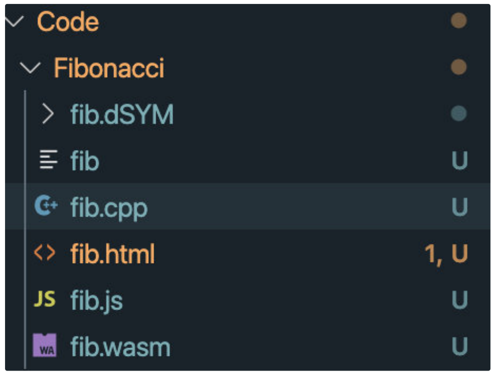


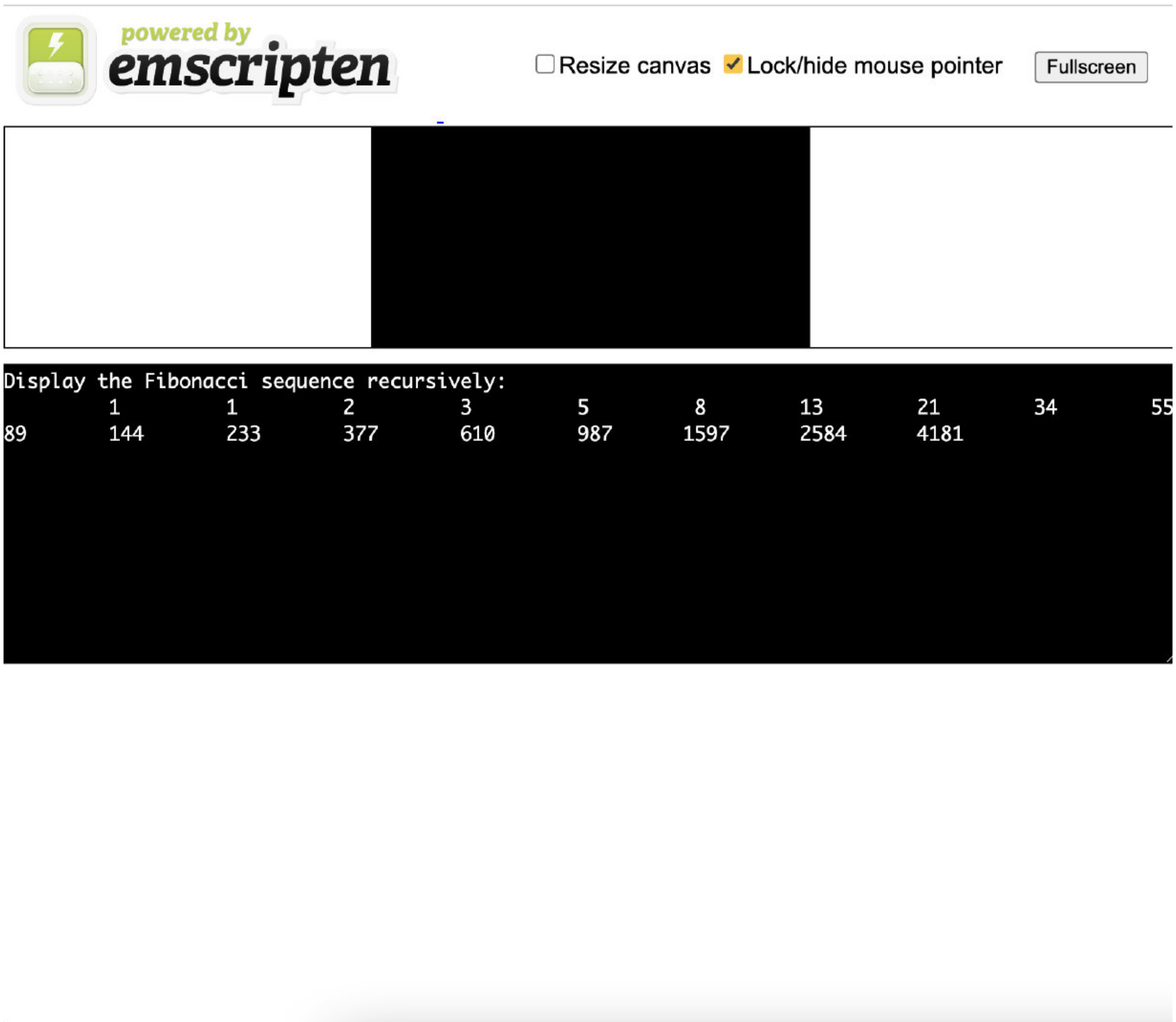
Test code base compiling

Then I used the same way and commands before to verify code base is functional and to ensure compiling is possible.

```
emcc fib.cpp -o fib.html
```

I get three new files which can run through Live Server.





Build code base to WebAssembly

Deleted those three transpiling files before. We now only compile WebAssembly file and JavaScript file with `emcc`.

First, I changed part of the code base to prepare a interface. And return 1 to prove that all work is done.

```cpp
#include <iostream>

#define N 30 // Output N Fibonacci numbers

using namespace std;

int fibArray[N];

int fib(int i)
{ // Recursively get Fibonacci number
    if (i < 2)
        return i == 1;
    return fib(i - 1) + fib(i - 2);
}

extern "C"
{
    int getFib()
    {
        // static int fibArray[N];
        for (int i = 1; i < N; i++)
        {
            fibArray[i] = fib(i);
        }
        return 1;
    }
}
```

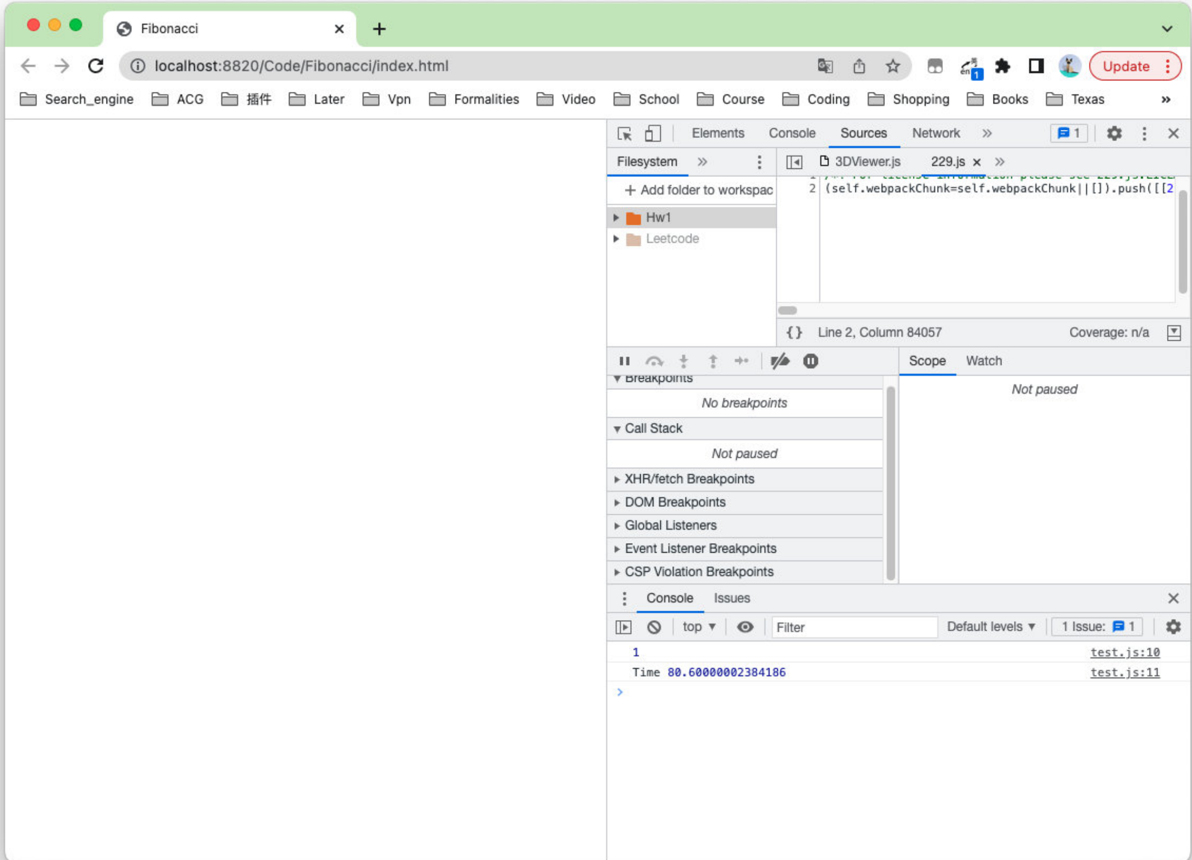
Then, compile it with this command below:

```
Fibonacci % emcc fib.cpp -o fib.js -s EXPORTED_FUNCTIONS=_getFib -s EXPORTED_RUNTIME_METHODS=ccall
```

Create an a test.js and index.html file to call getFib method and show the execution time on console.

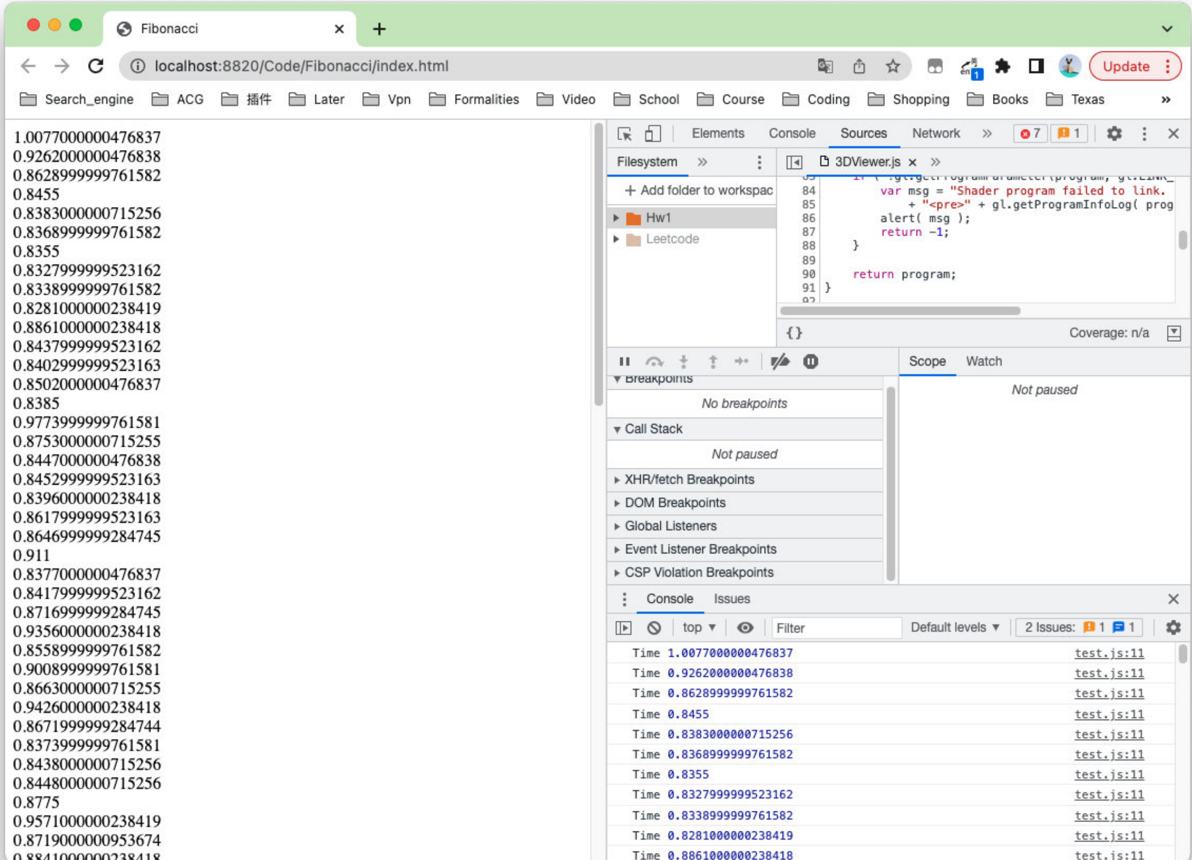
Code > Fibonacci > JS test.js > ...

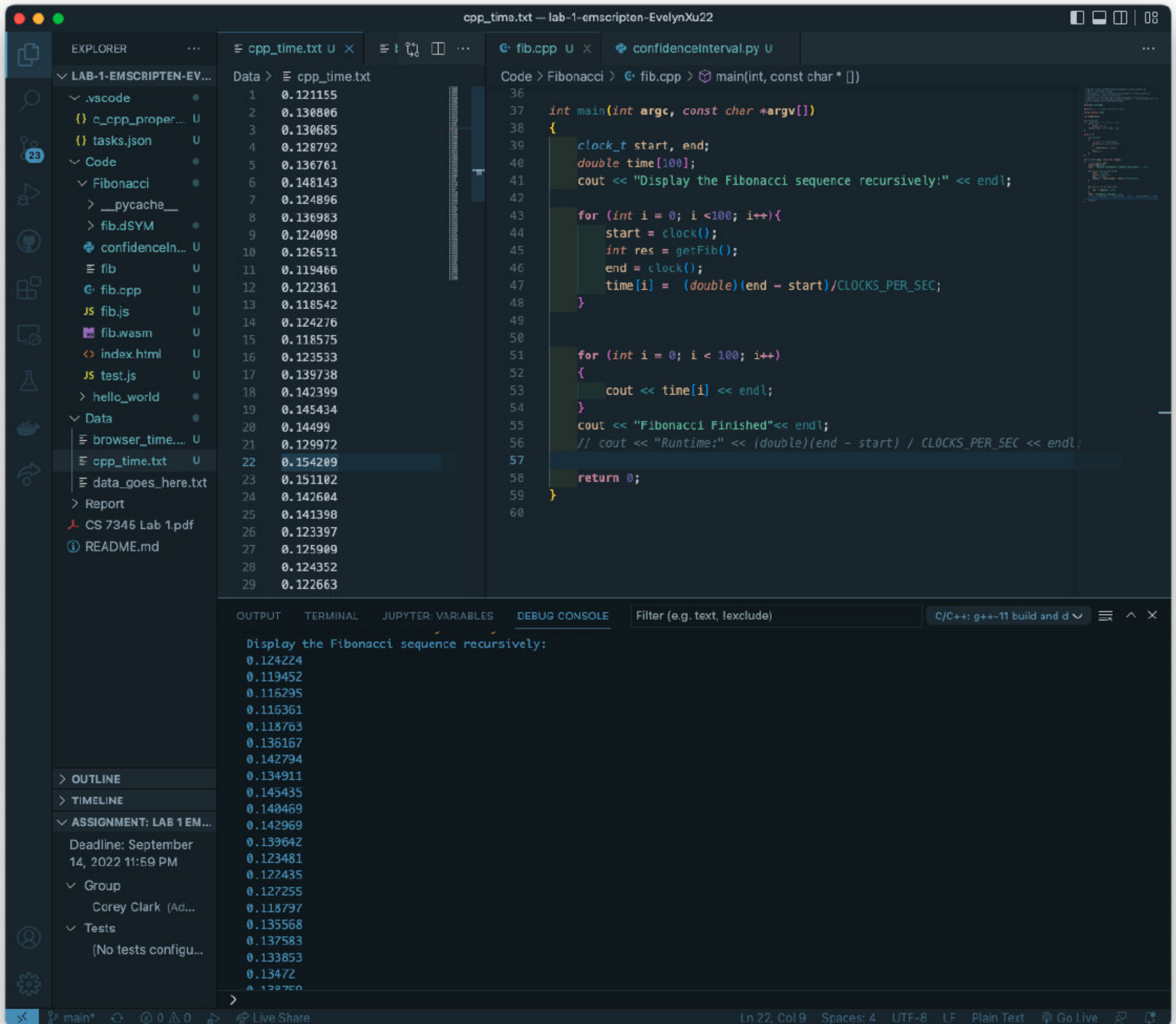
```js


// var Module = require("fib.js");
var test = Module.onRuntimeInitialized = () => {
    // for (var i = 0; i < 100; i++){
        var t0 = performance.now();
        var res = Module.ccall('getFib', 'number');
        var t1 = performance.now();
        asmTime = t1 - t0;
        console.log(res);
        console.log("Time", asmTime);
```



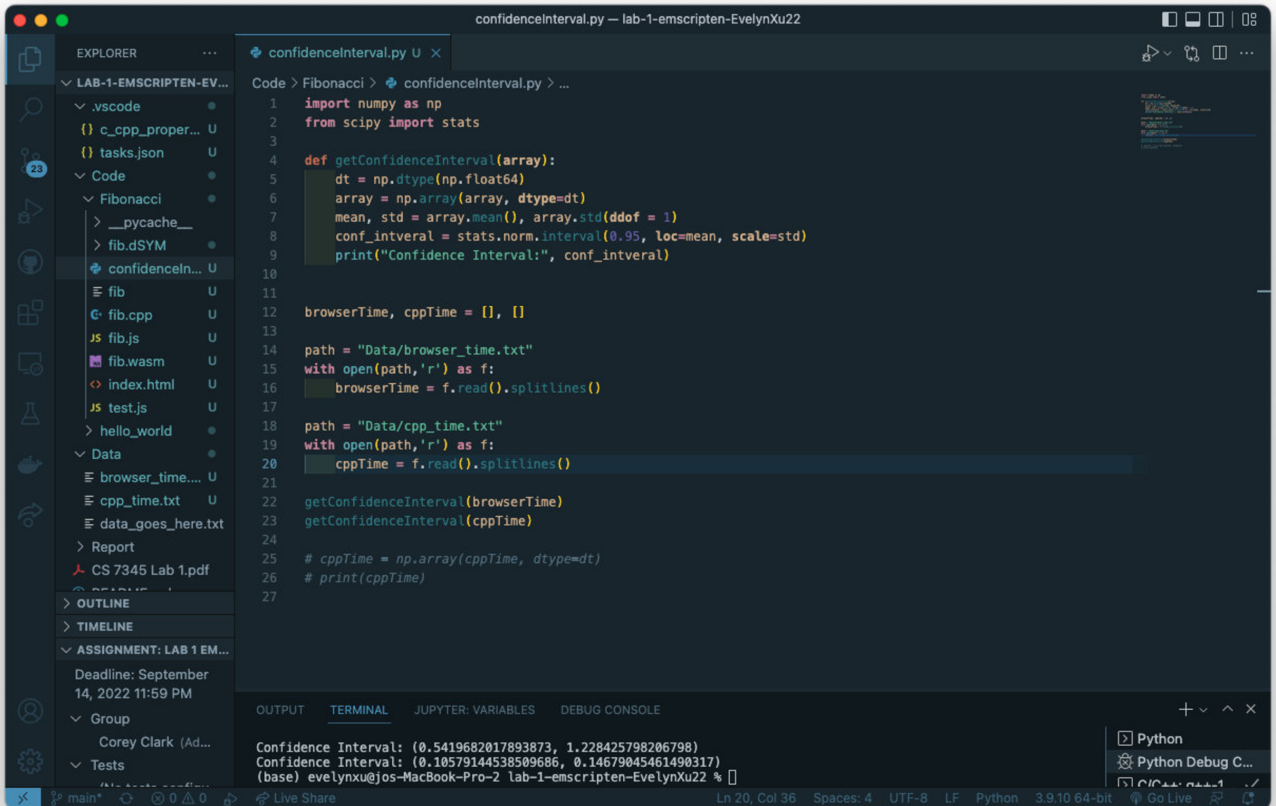
Compare execution time

In this step, I ran this code base 100 times both in C++ and in browser and save the results in `cpp_time.txt` and `browser_time.txt`.


Fibonacci
localhost:8820/Code/Fibonacci/index.html
Search_engine ACG 插件 Later Vpn Formalities Video School Course Coding Shopping Books Texas
Update
1.0077000000476837
0.9262000000476838
0.8628999999761582
0.8455
0.8383000000715256
0.8368999999761582
0.8355
0.8327999999523162
0.8338999999761582
0.8281000000238419
0.8861000000238418
0.8437999999523162
0.8402999999523163
0.8502000000476837
0.8385
0.9773999999761581
0.8753000000715255
0.8447000000476838
0.8452999999523163
0.8396000000238418
0.8617999999523163
0.8646999999284745
0.911
0.8377000000476837
0.8417999999523162
0.8716999999284745
0.9356000000238418
0.8558999999761582
0.9008999999761581
0.8663000000715255
0.9426000000238418
0.8671999999284744
0.8373999999761581
0.8438000000715256
0.8448000000715256
0.8775
0.9571000000238419
0.8719000000953674
Elements Console Sources Network
Filesystem
Add folder to workspac
Hw1
Leetcode
3DViewer.js
var msg = "Shader program failed to link.
+ "<pre>" + gl.getProgramInfoLog(prog
alert(msg);
return -1;
}
return program;
}
Coverage: n/a
Breakpoints
No breakpoints
Call Stack
Not paused
XHR/fetch Breakpoints
DOM Breakpoints
Global Listeners
Event Listener Breakpoints
CSP Violation Breakpoints
Scope Watch
Not paused
Console Issues
top
Filter
Default levels
2 Issues: 1 1
Time 1.0077000000476837 test.js:11
Time 0.9262000000476838 test.js:11
Time 0.8628999999761582 test.js:11
Time 0.8455 test.js:11
Time 0.8383000000715256 test.js:11
Time 0.8368999999761582 test.js:11
Time 0.8355 test.js:11
Time 0.8327999999523162 test.js:11
Time 0.8338999999761582 test.js:11
Time 0.8281000000238419 test.js:11
Time 0.8861000000238418 test.js:11



Next, I wrote a python-based code for running time analysis and confidence interval calculation, named `confidenceInterval.py`. This function read the results were saved before and calculate confidenceInterval with `numpy` and `scipy`.



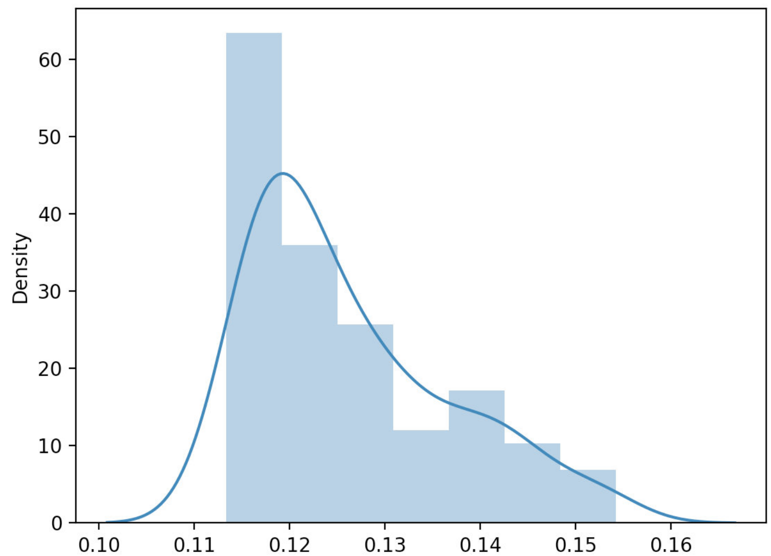
As the screenshot before, we can get 95% confidence interval:

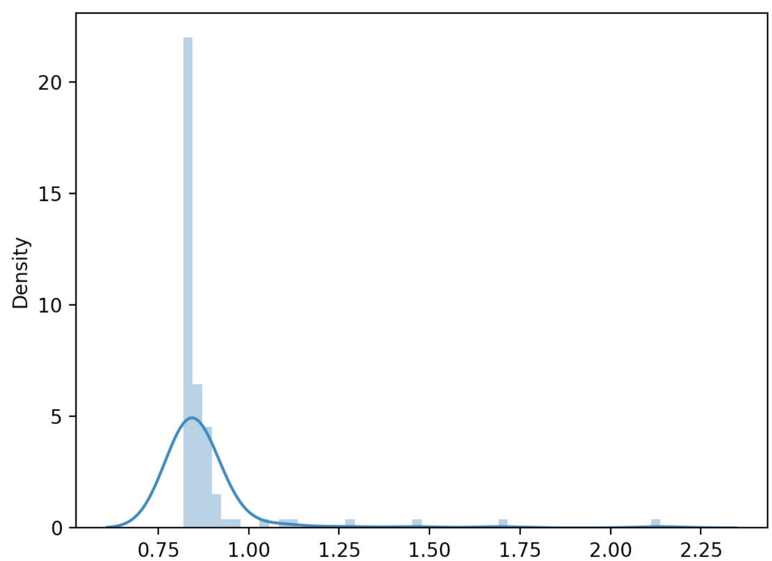
- native (C++): (0.10579144538509686, 0.14679045461490317)
- WASM-based code: (0.5419682017893873, 1.228425798206798)

According to these execution time, we can see WASM-based code takes about 10 times longer to execute than the native code.

However, the data distribution does not conform to a normal distribution. The data distribution is shown below.

Nativa Code Runtime Distribution:





WASM-based Code Runtime Distribution:

And we can also see that there are some unusually large values. In my analysis, the reason for such large differences could be that there are times when the memory is over-occupied by other processes during the running of the program, resulting in a few particularly irregular runtime values that are particularly large.

Since these problems may have caused errors in the mean and standard values, we can learn that the confidence intervals obtained can be considered statistically insignificant.

Improve performance

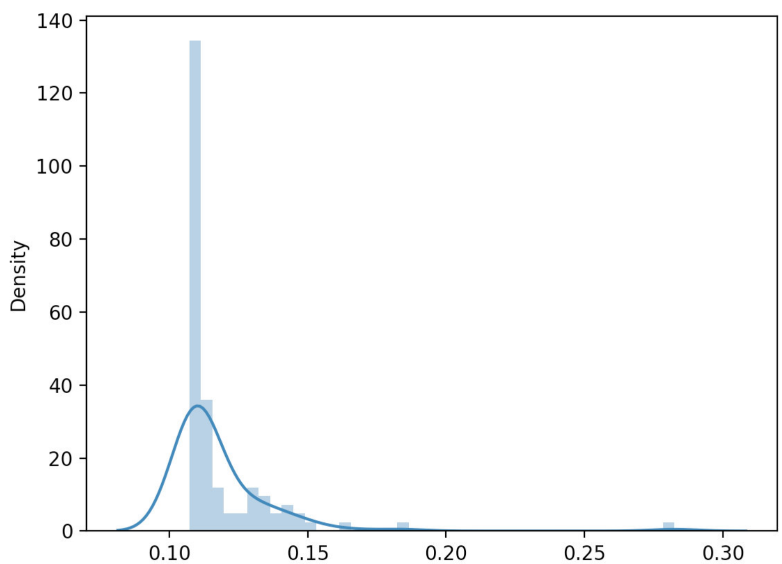
We can optimize by specifying the optimization flags, which are:-O0, -O1, -O2, -Os, -Oz, -O3. Here I choose -O2 level optimization for compiled.

```
Fibonacci % emcc -O2 fib.cpp -o fib.js -s EXPORTED_FUNCTIONS=_getFib -s
EXPORTED_RUNTIME_METHODS=ccall
```

After running this function 100 times and saving the output to `browser_O2_time.txt', we can get its 95% confidence interval in the same way as before.

- native (C++): (0.10579144538509686, 0.14679045461490317)
- WASM-based code: (0.5419682017893873, 1.228425798206798)
- WASM-based code (-O2): (0.07599345381156607, 0.16062854619367917)

WASM-based Code (-O2) Runtime Distribution:



Although its distribution is also skewed and can be considered statistically insignificant, we can still learn that it is actually much faster than the WASM-based code without any optimization, and its performance is even close to that of the native code.