Problem Set 3

Deadline: October 23, 2023 22:00 Hrs

Important Instructions

This assignment is exclusively coding based. Once you are done with it, **submit a PDF on gradescope** with only the following:

`"My final commit is: <commit-hash> <commit-msg>"` of the commit you want us to grade.

- You don't need to add any code to print stuff. We have done that for you *already* to avoid stupid autograding issues.
- Only fill in what's specifed later in this doc. Read the comments in the code. You need not write any extra functions beyond what's already given.
- Lastly, since we have included a large chunk of starter code it is important that you go through all the pre-written functions to get a good grasp of how to code up the solutions.

0. Skill Check

Before proceeding, please ensure you are already familiar with performing basic tasks using a `terminal`:

- What terminal commands and flags are
- What directories are and what a `path` is
- Navigating directories using the terminal
  - Finding the current working directory using `cwd`
  - Creating a directory using `mkdir`
  - Removing directories and files using `rm` and the `-rf` flag
  - Changing directories using `cd`
- Have `gcc` and `make` installed and are familiar with basic usage.
  To review: See setup portion of Assignment 1.
- How an IDE (like VS Code) provides a simple GUI for the above tasks
- How to open a folder in an IDE and work with it
- Understand pointers, `malloc`, `calloc`, `typedef` and `structs` in `C`
- Understand parameters, function/method signatures, `pass by value` and `pass by reference`.

1. Link to the Assignment

Accept the assignment using this link: https://classroom.github.com/a/M5z2sDqE

2. Structure of the Repository

Clone your assignment repo locally, open it in an IDE (like VS Code / Sublime Text / CodeBlocks / whtv suits you) or a cool Text Editor (like Vim) and start working on it. You have been provided with boilerplate (aka starter) code.

- Folders pertain to **one** functional component of assignment.
- For each question, find the folder with the relevant name below.
- Your task is to fill **only** those functions mentioned below on a per-question basis.
- Look for `// TODO` in the files.

Each folder (corresponding to a question) will contain:

- some `.c` files (which contain C source code)
- some `.h` files (which are called header files and help share source code)
- **one** `main.c` file (which is used to test and run the code)
- and a `Makefile` (which is helps simplify the compilation process)

This has been done to help autograde the relevant components easily. Do not worry about the `main.c` and `Makefile`s in these folders. You are **not** supposed to touch them. If you make changes to these files, the *autograder* will **break**.

Importantly, peruse the `.h` carefully. They contain important declarations for `struct`s and functions that you are supposed to use (or implement). It is through these *header* files that the code in the corresponding `.c` file is made visible to other files. This is why you'll see a number of `#include "someheader.h"` across the codebase.

If you notice carefully, functions *not in the header file* are *not visible* outside of the corresponding C file.

For e.g.: `set_r()` in `hashing.c` cannot be used in any other file even if you `#include "<path_to_hashing.h>"` in that file.

Keep this in mind while figuring out which functions are helpers *inside* a file, and which others provide an interface to your data structure *outside* the file.

The repository is structured as follows:

- `prng/`: Contains the Pseudo-Random Number Generator (DO NOT TOUCH)
- `helpers/`: This folder contains the source code (like a library of sorts) that provides you with the following functions (DO NOT TOUCH):
  - `randomArray()`: Creates a random array of size `n`.
  - `duplicateArray()`: Creates a duplicate of a given array by allocating memory for it. Similar to deepcopy in python. Do NOT forget to free this array after you're done.
  - `printArray()`: Prints an array to the terminal.
  - `sortArray()`: Sorts an array
- `tests/`: A folder containing expected outputs so that you can test locally, if you want.
  - `expected/`: Contains expected outputs for each question in a `.txt` file.
- `hashing/`: Contains all the files related to hashing (**Question 1**). This will be used in all subsequent questions.
  1. `hashing.c`: Implement the dot-product hash family. CODE GOES HERE
  2. `hashmap.c`: Implements a simple hashMap using the `hashFunction()` implemented by you. Uses linear probing. (DO NOT TOUCH)

- `isomorphic-digits/`: Code for **Question 2**
  - `isomorphic.c`: CODE GOES HERE
- `replace-by-rank/`: Code for **Question 3**
  - `replacebyrank.c`: CODE GOES HERE
- `bloom-filter/`: Code for **Question 4**
  - `bloomfilter.c`: CODE GOES HERE

3. How to write your code

Again, **DO NOT TOUCH any of `main.c` files. Fill ONLY the relevant functions.**

- You can refer to the function signatures in corresponding `.h` files to understand the datatypes of the `parameters` and the `return` type of the function.
- You are free to use the functions provided to you as-is. Do keep in mind the function signature while passing arguments and assigning its return value to a variable.
- *DO NOT* change any function signatures, including those that you are required to implement. This will cause you immeasurable pain during compilation.

Work on **Question 1** first, because all the other questions are dependent upon your work there.

4. Compiling and Running

Important: We have noticed insane compute usage for the GitHub autograder – You are **NOT** supposed to use the autograder to do the compiling for you. People have submitted code with infinite loops / input that was never given. **Do not** push to GitHub if your code doesn't compile or gives no output when run. Continued abuse of privileges made available shall lead to termination – we shall switch to a local autograder and real-time results of your work will become unavailable. Please use your discretion accordingly.

Follow the steps below to compile and test your work:

- Make sure you are in the directory relevant to the question you're working on.
- Type the following in the terminal to compile your code for that question.

```
make <executable_name>
```

- After **successful compilation** your code will be tested using the `main.c` file *in that folder*. It contains the `main()` function. Relevant tests have been written here, you do not need to edit this.
- Assuming you're in the relevant directory, **run** the executable obtained after compilation.

For Mac/Linux:

```
./<executable_name> [args_if_required]
```

For Windows:

```
.\\<executable_name.exe> [args_if_required]
```

- If you run your program with the incorrect number of arguments, it will print the correct usage. Follow that. The relevant usage is mentioned below each question (run appropriately for Windows).
- Check your program's output for correctness. You will likely not face issues with output formatting since the print functions have been written for you.

- If you're facing issues, and want to create a fresh executable, you can:

```
make clean
make <executable_name>
```

- Note that Windows handles some UNIX commands differently. You may get a Makefile error like below, which simply indicates that the `rm` command didn't work on your (Windows) machine. Look carefully, you'll see `Makefile:<line no.>` telling you the line no. in the Makefile associated with the error. Do not worry about it. You should be fine as long your work compiles using `gcc` or `clang` and produces correct output.

```
process_begin: CreateProcess(NULL, rm -f main.o bloomfilter.o ../hashing/hashing.o ../hashing/hashmap.o ../helpers/helpers.o ../
prng/mt19937-64.o, ...) failed.
make (e=2): The system cannot find the file specified.
make: *** [Makefile:16: bloomfilter] Error 2
```

- If `make clean` doesn't work, you may (equivalently) manually delete:
  - the executables, i.e, `.exe` files
  - the object files, i.e, `.o` files
  - To do this on Windows' Powershell, execute:

```
del *.exe *.o ..\hashing\*.o ..\helpers\*.o ..\prng\*.o
```

5. Testing Locally for Correctness

We will be matching **exact** output for autograding. The test command for each question is given below (the `./` will be different for Windows). You can redirect your output to a text file and `diff` with `.txt` files provided in `tests/expected/`, or simply open these to see exactly what the result should be.

1. `./testhash 5 10 9`
2. `./isomorphic`
3. `./replacebyrank 2181 15`
4. `./bloomfilter 9 filterparams.txt`

6. Debugging

a. Easy Way - `printf()`

Sprinkle `printf()` wherever you think your program could have gone wrong. Often it helps to add a printf() before and after a block to check for issues.

b. Using a debugger - `gdb` or `lldb` (for Apple Silicon)

(**Optional**) If you are comfortable using a debugger (now is a great opportunity to learn, if not!), navigate to the relevant directory and run:

```
make debug
```

This will create an executable with debug symbols called `<executable_name>-debug` that you can then attach to your debugger and go crazy with. Feel free to drop by during OH if you'd like a walkthrough!

Once you're done debugging, follow the steps above to create a fresh executable.

Write your code for one question. Compile. Resolve errors. Run. Move on.

Question 1

Universal Hashing - 80 points

Assume your that your *universe of elements* U comprises of positive (unsigned) integers less than 50,000. Implement the following dot-product hash family:

$$h(k) = a.k \bmod m$$

- `m` is a prime number
- `a` is a randomly generated vector (which determines the hash function)
- `k` is the `key`

Review the dot-product hash family here, if needed.

Look through `hashing.c` (and `hashing.h`). Each function *blackboxes* an important step in your overall implementation. Think about how these pieces fit together: which functions are called inside which others?

Implement the following features of your hash function in `hashing.c`:

1. `setup()`: Sets up a hash function with a randomly generated vector of digits in base `m` (which is prime). This vector is stored as an array of (unsigned) `int`s.
2. `destructHash()`: Deallocates memory allocated to the components of the hash function.
3. `hash()`: Return an `index` between `0` and `m-1` (inclusive) given a `key`. This is called the *"hash"* of the `key` under the current hash function.
4. Fill in all the empty functions in the process and use them to perform the necessary tasks when implementing the above mentioned features of your hash function.

Hint: Find out the difference between `malloc()` and `calloc()`. Think about where you might need it.

A *HashMap* has been implemented for you in `hashmap.c`. The accuracy of tests is ensured by the platform-agnostic Pseudorandom Number Generator located inside `prng/`, so you need not worry about tests failing due to randomness. This ensures that the tests are deterministic. If your hash function is implemented correctly, it will pass all autograder tests.

```
Usage: ./testhash <seed> <table_size> <array_length>
```

`<array_length>` is the length of the randomly generated array that will be used to test your hash function.

Only proceed further once you're done with this question.

Question 2

Isomorphic Numbers - 30 points

Check whether the digits of two given numbers are isomorphic. Assume that two numbers are isomorphic if there exists a one-one correspondence between the digits of the two numbers. By default, numbers of unequal length are NOT isomorphic.

Examples: `a = 458109004` and `b = 197356551` are isomorphic.

$$digit\ in\ a \rightarrow digit\ in\ b$$

$$0 \rightarrow 5$$

$$1 \rightarrow 3$$

$$4 \rightarrow 1$$

$$5 \rightarrow 9$$

$$8 \rightarrow 7$$

$$9 \rightarrow 6$$

`p = 1234567890` and `q = 0987456231` are isomorphic.

`k = 1234556777890` and `l = 0987495660231` are **not** isomorphic since there is no unique correspondence for digits `5` and `7` in `k` to the digits in `l`.

There need not be a pattern in the way the digits are mapped, as long as each unique digit maps to some other unique digit.

Your output should look like:

```
12345 and 54312 are isomorphic.
67980 and 67890 are isomorphic.
132477 and 123477 are isomorphic.
458109004 and 197356551 are isomorphic.
1234567890 and 9087456231 are isomorphic.

2345654 and 1453 are NOT isomorphic.
2345654 and 2435663 are NOT isomorphic.
234556777890 and 987495660231 are NOT isomorphic.
```

- Use the **hash function** you created in Question 1 for this.
- Use the **hashMap** provided to you to implement your solution.

Usage: `./isomorphic`

Question 3

Replace by Rank - 30 points

You are given a set of numbers

$$n_0, n_1, n_3, ..., n_k$$

Assume there are no duplicates in the input array. Replace each number by their rank.

For e.g:

```
Input : 123, 47, 698, 7, 80
Output: 4, 2, 5, 1, 3
```

Note that ranks are numbered starting from `1`. Your `output` should be exactly of the above form. That is, for an input of 4 numbers:

```
Output: Rank, Rank, Rank, Rank
```

TODO:

- Code up: `replace_by_rank()` in `replacebyrank.c`
- Ensure you've setup and `free()`d all the data structures you've used.
- Test your work with by running it with the following cmd-line args:
  1. `seed` = 2191, `array_length` = 12
  2. `seed` = 2181, `array_length` = 15
  3. These particular seeds provide arrays of the given sizes with no duplicate elements.

Again, make use of `hashFunction` and `hashMap`.

Usage: `./replacebyrank <seed> <array_length>`

Question 4

Bloom Filter - 60 points

Create a bloom filter with the following components:

1. `k` distinct hash functions: Create `k` instances of $h(x)$ from `Question 1`, each one with a different `m`. These different `m`s are passed as an array called `sizes` when creating a new bloom filter.
2. A bit array of size `fs` aka the `filter_size`

The filter should support the following operations:

1. `createFilter()`: Initialise a new bloom filter and the necessary components.
2. `insertFilter()`: Update the state of your filter after *seeing* a new element `x`.
3. `searchFilter()`: Return whether a given `x` is ***definitely not*** in the filter or ***probably*** in the filter.
4. `printFilter()`: Print the current state of the filter.
5. `freeFilter()`: Deallocate all the memory associated with a bloomfilter, including its hash functions.

Closely refer to the functions provided, they will help you structure your thought into code.

Usage: `./bloomfilter <seed> filterparams.txt`

Hints:

- The `hashFunction` returns an index based on its `m` which is prime. How would you ensure that the indices are within range for your `filter`?
- Think about how you would represent a data structure to keep your `k` hash functions in C.

Tips:

- Carefully peruse the `structs` given in the boilerplate code. It will make a lot of things clear.
- These are **not** difficult to code up. It's way easier than it looks. However, you must be clear with what's going on in each function to be able to write it up.
- Use assertions, i.e, `assert(<boolean condition>)` to check that inputs to functions are valid. You will find some examples of these in the code given to you.