Project 4

<div style="border:1px solid">Start Assignment</div>

Due Apr 24 by 11:59pm   **Points** 10   **Submitting** a file upload   **File Types** zip

Available Apr 7 at 2pm - Apr 24 at 11:59pm 17 days

Project 4: Proof-of-Work Generator/Validator

This project asks you to write a set of programs to create and to validate proof of work values.

Sample files and sample outputs: **p4-samples.zip** ↓
(https://rutgers.instructure.com/courses/160949/files/24006685/download?download_frd=1)

Introduction

Proof-of-work was created as a mechanism to demonstrate that the creator of the proof-of-work had to put in a substantial amount of computing effort while making it efficient for a recipient to validate this.

▶

In Bitcoin, proof of work serves two functions:

1. It makes creating a block with a valid block hash so difficult that it is extremely unlikely that two bitcoin nodes would propose a block at the same time.

2. It takes so much computation to compute the proof of work to create a valid hash for one block that the amount of computing power an attacker would need to modify a sequence of hash pointers for many blocks to change an old transaction is not feasible.

This assignment contains two parts:

1. In part 1, you will implement a program that computes a proof of work for a file. You will specify a difficulty level on the command line and the program prints a set of headers as output.

2. In part 2, you will implement a program that validates the headers produced in part 1. If the proof of work is incorrect or the hash is incorrect then the validation will fail.

Environment

This is an individual project. All work should be your own except for the referenced code for the hash function.

You may use Go, Python, Java, C, or C++ in your implementation.

You should be able to implement this on any platform but you are responsible to make sure that the program runs on Rutgers iLab systems with no extra software.

Background

Hashcash was an idea to reduce the amount of email spam. It would do this by requiring the sender to solve a difficult puzzle before sending a message.

The sender would then provide proof that this puzzle was solved and messages would be rejected if such a proof was missing or in valid. The proof, called a *stamp*, was provided as a header in the mail message.

Someone using hashcash might spend several seconds creating the stamp. That's not disruptive for someone who is sending a few messages but is essentially impossible for a spammer who needs to generate stamps for millions of messages. Because the puzzle uses the content of the mail message and the recipient's address, the spammer would need to spend many years of computing time to send a million messages as the stamp would need to be computed for each recipient and is unique to each

▶ age.

The solution to the puzzle needs to be verified efficiently by receivers, so we need a puzzle that's difficult to solve but easy to verify.

The idea of hashcash was adopted by Bitcoin and many other cryptocurrencies to enable participating systems provide evidence of Proof of Work when adding a new block to the bitcoin blockchain.

The puzzle

Hashcash had to create a puzzle that was a function of the message, was sufficiently difficult to compute, but was efficient to verify. This aligns closely with properties of one-way functions. Cryptographic hashes are one-way functions that provide fixed-length output.

For example, here's a SHA-256 (256-bit SHA-2) hash of the text 'The grass is green':

```
f3ccca8f3852f5e2932d75db5675d59de30d9fa10530dd9855bd4a6cd0661d8e
```

It takes only a few milliseconds to compute this. The inverse function, finding some text that would produce this hash, requires a brute-force search. You would need to try hashing a huge amount of

different messages to find one the produces the same hash.

For example, Google attacked the SHA-1 hash but only by trying over 9×10^{18} hashes. The attack required 12,000,000 GPU years of computing power. A stronger hash function, such as SHA-2, would take far longer. Clearly, reversing a hash is far too difficult a problem to use as a puzzle.

An easier puzzle

We can make the puzzle easier. Suppose we come up with some text, W, that we append to the message *M*. When we hash the concatenated message, *hash(M||W)*, the resulting hash value will have a certain property.

For example, this was our SHA-256 hash of 'The grass is green':
f3ccca8f3852f5e2932d75db5675d59de30d9fa10530dd9855bd4a6cd0661d8e

If we look at the first bytes in binary, we see the bits are 1111 0011 1100 … . Suppose we want to add some text to the message so that the resulting SHA-256 hash will be a value whose first 6 bits are 0.

There is no way to predict how to create a message that will result in such a hash. The only thing we can do is try different combinations, suffixing the message with different values of *W*.

This particular challenge turns out not to relatively easy and we can solve it in a few milliseconds. For example, if we add the letters '[hB]' to the text, we get this hash:

> ▶   cdcf382f41bd88c65147ecf0328fc0258b0a49417c568891962e631d4c2d

The binary value starts with 6 zeros:

```
0000 0010 0100 0011 ...
```

Adaptive complexity

We can adjust the difficulty of the puzzle by altering the number of leading 0 bits that we need to find in the resulting hash. This difficulty will be only an average. Sometimes we might get lucky and find a suffix that produces the desired result quickly. Other times it will take longer.

Here's a table of some of the suffixes I came up with for the text 'The grass is green' for different difficulty levels, where *d* is the number of leading 0 bits in the resulting hash when the string *W* is appended to the text.

Difficulty, d	Suffix, W	Iterations	Time (s)
9	[T]	19	0.000023

Difficulty, Suffix, d	W	Iterations	Time (s)
19	`@J9B`	597,172	0.2138
23	`/8hV`	6,759,807	2.4135
24	`5#PgD`	74,279,789	26.586
27	`uwT2G`	145,358,750	51.930
32	`6j8wCC`	2,966,002,502	1,065.6
34	`fbI3xN`	19,966,156,427	7,220.0
36	`12iP#X`	34,835,138,909	12,532

In this example, I found a string suffix that results in a hash with 19 leading 0 bits by testing around 600,000 variations of suffixes in less than a quarter of a second. However, finding a suffix that results in a hash value with 23 leading 0 bits took testing almost almost seven million suffixes and took almost 2.5 seconds.

The puzzle gets difficult quickly. Finding text that would produce a hash with 32 zero bits required testing almost three billion different suffixes and took a bit over 17 minutes. To get 34 leading zeros, I had to test close to 20 billion suffixes and that took two hours on my M1 Mac Mini. Producing a hash with 36 leading zeros required testing over 34 billion suffixes and about 3.5 hours of time. Note that I was using non-optimized code and hash computations can be performed a lot faster within a GPU (graphics processing unit) but the difference in complexity remains the same.

Size-invariant difficulty

The time to compute a hash is longer for large messages; we need to iterate over more bytes to compute the hash. To make the difficulty not be a function of the message length, we can modify the puzzle to one where we append suffixes to a hash of the message and then hash the resulting message to see what we get.

The average difficulty of the problem now is simply the value d, which defines the number of leading 0 bits in the hash result. The actual time it takes to find the right suffix W will depend on:

- Luck: we might get lucky and test some strings early on that will provide the hash result we want. We may get lucky occasionally but not most of the time. The law of averages will play out.
- The speed of your computer and the quality of your code. We can speed up hashes greatly by using GPUs or custom processors as people do with bitcoin mining, but we won't do that in this assignment.

- Mostly, the difficulty depends on the value of *d*, the number of leading zero bits we want in the resulting hash. Searching for a hash that produces a greater number of

Proof of Work

The suffix that was found — the string *W* — that, when appended to the message, produces the desired hash output, is called the Proof of Work.

It is a short string that we can provide to prove that we did the work to find this value that produces the desired hash.

The proof of work is very efficient to test. For instance, it took my program over 19 billion hashes to find that we can append the string `fbI3xN` to the text `The grass is green` so that the first 24 bits in the resulting hash will all be 0. This string is the proof of work. You can verify the proof of work by computing one hash and looking at the resulting value. It will take you only a millisecond.

Your assignment: part 1

Your assignment is to write a program called `pow-create` that creates a proof of work string for a given file:

```
create nbits file
```

The work is the search for a string that, when suffixed to a hash of the given file (*file*), will result in a SHA-256 hash (256-bit version of the SHA-2 hash) that contains a certain number (*nbits*) of leading 0s.

For example, suppose we have the following text in a file called walrus.txt:

```
The time has come, the Walrus said,
To talk of many things:
Of shoes — and ships — and sealing-wax —
Of cabbages — and kings —
And why the sea is boiling hot —
And whether pigs have wings.
```

We can use the *openssl* command on a macOS or Linux system to find its SHA-256 hash:

```
$ openssl sha256 < walrus.txt
66efa274991ef4ab1ed1b89c06c2c8270bb73ffdc28a9002a334ec3023039945
```

The hash starts with `66ef`. The hex digit 6 is `0101` in binary, so we currently have one leading zero bit in this message.

To create a proof of work string with a difficulty of 20 (at least 20 leading zero bits), we run the command:

```
$ pow-create 20 walrus.txt
File: walrus.txt
Initial-hash: 66efa274991ef4ab1ed1b89c06c2c8270bb73ffdc28a9002a334ec3023039945
Proof-of-work: rftE
Hash: 000005ca35310f45d7ef2b28753a74cf410734b4a5930247d15128d23e419ca0
Leading-zero-bits: 21
Iterations: 1467959
Compute-time: 1.0712
```

Your **pow-create** command will:

1. Create a SHA-256 hash of the specified file.
2. Convert it to a printable hex string (matching the string produced by the *openssl* command).
3. Pick a string that's a potential proof of work value.
4. Create a hash of the string representation of the hash in (2) concatenated with the potential proof of work string in (3).
5. If the hash doesn't start with at least *nbits* zero bits (the number supplied by the first argument of the command line) then go back to step (3) and try a different suffix.

For this example, it took almost 1.5 million hashes with different variations of suffixes to find a hash that starts with 20 zero bits (in this particular case, the first hash we found that started with at least 20 leading 0 bits actually contained 21).

The string that we came up with is `rftE`. This is presented as the proof of work in the `Proof-of-work` header.

Output

Your program will print results to the standard output (stdout) in a standard header format as used by mail headers or HTTP. This is one name-value item per line with each line containing the header name, a colon, one or more spaces, and the value. A header item shall not span multiple lines and the output will not contain blank lines.

The headers your program must produce are:

File:
   The name of the file.
Initial-hash:
   The SHA-256 hash of the file (as a printable hex value)
Proof-of-work:
   The printable string that is your proof of work.
Hash:
   The SHA-256 hash of the original string concatenated with the proof of work.

Leading-zero-bits:

The actual number of leading 0 bits in the hash you computed. This value should be greater than or equal to the number requested.

Iterations

The number of different proof-of-work values you had to try before you found one that works.

Compute time:

How long this process took, in seconds (including decimal seconds if appropriate).

Testing your proof of work

You can easily test your program's result against the data produced by the *openssl* command

Run your pow-create command:

```
$ ./pow-create 20 walrus.txt File: testfiles/walrus.txt
File: testfiles/walrus.txt
Initial-hash: 66efa274991ef4ab1ed1b89c06c2c8270bb73ffdc28a9002a334ec3023039945
Proof-of-work: rftE
Hash: 000005ca35310f45d7ef2b28753a74cf410734b4a5930247d15128d23e419ca0
Leading-zero-bits: 21
Iterations: 1467959
Compute-time: 1.0712
```

The only header we care about here is the `Proof-of-work` value, `rftE`.

We can check our hash with the one *openssl* produces by running the *openssl* command with the
▶ `6` argument. This value should match the `Initial-hash` header:

```
$ openssl sha256 <walrus.txt
66efa274991ef4ab1ed1b89c06c2c8270bb73ffdc28a9002a334ec3023039945
```

Then we append the proof of work to that hash string and find the sha-256 hash of this proof of work string concatenated with the hex string output of the original hash:

```
$ echo -n '66efa274991ef4ab1ed1b89c06c2c8270bb73ffdc28a9002a334ec3023039945rftE'|openssl sha256
000005ca35310f45d7ef2b28753a74cf410734b4a5930247d15128d23e419ca0
```

We can see that the resulting hash starts with five zeros followed by a 5 (which is 0101 in binary). We have a hash that has 21 leading zero bits, which is at least as good as the 20 we wanted, so the proof of work is valid.

Hints

This is a super-short assignment. You won't write a SHA-256 hash function from scratch. In python, you can use `hashlib`. In Java, you might use the `java.security.MessageDigest` package. In other languages, you may need to find source code that computes a hash.

If you're using source code for a SHA-256 function, do NOT submit multiple files for an entire crypto library package. The code for computing a SHA-256 hash is small. Submit only the source file you need for that function and cite your where you got the source in your comments.

If you are using a hash function that needs to be compiled, your Makefile needs to have instructions on how to build all the code.

Make sure your hash function works on the iLab systems.

Test that your hash function produces the same results as openssl. You need this for testing suffixes. As with writing cryptography functions, the recipient needs to be able to verify your work with their own code.

I'm not telling you how to find suffixes for proof-of-work values. There are various ways you can do this and you should figure out something that works. Keep your proof of work as **printable 7-bit ASCII text** (e.g., no extended characters such as ü, ñ, or é) so they can be presented in the headers. This means no whitespace characters. To make testing easier, do not use quote characters as part of the string either. You can iterate through a select character set, use base64 encodings, or anything else. Just be prepared to extend your proof of work from one byte to several as the search gets more difficult.

Watch your workload

As shown in the table above, finding long sequences of leading zeros can take an extremely long time –

▶

s exponentially more difficult. That is why bitcoin miners use data centers filled with custom ASIC ~~processors~~ essors that try trillions of hashes per second.

You might want to set a threshold on the number of suffixes you try so your program won't run for an unbounded time. For example, you might set it to a billion or so, especially on a Rutgers system. For your PC, you can set it to a few tens of billions.

Test your code with small difficulty levels. If you can generate proof of work strings ranging to a difficulty of, say, 20 bits, then your code will likely work fine for longer amounts of leading zeros.

You don't want to cause a strain on computers if you're using shared Rutgers systems … or get your account locked because you used up your CPU quota.

Making it real

If you were going to use this type of proof of work system for real, you might make a few changes. You would not bother turning the hash into a text string but rather hash the 256-bit binary hash. Bitcoin hashes the block header, which includes the proof of work value, the Merkle tree hash for transactions in the block, and a few other fields.

We're using the text representation of the hash only to make testing and printing more convenient. It does not diminish the complexity of the work and, in fact, makes the hashing somewhat longer since we are hashing strings that are 64 bytes (512 bits) plus a short but variable-size proof-of-work suffix string.

The header for the assignment prints data that is not necessary. You will not care about the hash values since anyone can recompute them. The only important fields are the proof of work value and the number of leading zero bits that that the result promises to have (and even this latter value can be made optional since you can compute the hash and reject shorter results).

Depending on the application, you would likely use longer difficulty values that would make it not feasible for an attacker to recompute. If this is used for an application such as email (e.g., hashcash), a value that can be computed within a few seconds is fine to discourage spammers. If this is used in a blockchain to support distributed logging, for instance, you'd use a much longer value. Bitcoin, for instance, typically requires around 74 leading zeros in its hash.

Your assignment: part 2

The second part of the assignment is a verifier. It is called **pow-check** and has the following syntax:

```
pow-check powheader file
```

The command is provided a file the headers (generated by the **pow-create** command and a file with the ▶ ial message. It validates the headers against the file.

These are the tests it performs:

- It checks the value of the `Initial-hash` in the header. This is the SHA-256 hash of the message.

- It computes the hash of the initial hash string concatenated with the `Proof-of-work` string in the header. This value should match the `Hash` header

- Finally, it checks that the number in the `Leading-bits` header **exactly matches** the number of leading 0 bits in that hash header.

The result of pow-check will be a single line with the message pass or fail. If all conditions passed, simply print a `pass` message. If any tests failed, specify which of these tests failed before printing a line the `fail` message. The final line of your output contains this single pass/fail result. Previous lines *may* identify checks that passed and *should* identify checks that failed.

Sample output

You can – and should – do your own tests using **openssl** but I've supplied you with **sample files** ⬇ **(https://rutgers.instructure.com/courses/160949/files/24006685/download?download_frd=1)** that contain

header output for various input files. You should produce headers similar to these and your **pow-check** program should successfully validate all these headers.

For example:

```
 $ ./pow-check headerfiles/abc.pow-20 testfiles/abc
PASSED: initial file hashes match
PASSED: leading bits is correct
PASSED: pow hash matches Hash header
pass
```

You should then modify some aspects of the header or test against the wrong file to force pow-check to fail.

Example 1: wrong file

If you provide a file that was not used to generate the header, the initial hashes will not match. You need to detect that the initial-hash value does not match the hash of the file.

```
$ ./pow-check headerfiles/abc.pow-20 testfiles/alice.txt
ERROR: initial hashes don't match
       hash in header: 1010a7e761610980ac591359c871f724de150f23440ebb5959ac4c0724c91d91
       file hash: 4c2824e599717cc70abe29345d326c0ad4c4564b79bab6570eb0766834829d68
ERROR: incorrect Leading-bits value: 20, expected 3
ERROR: pow hash does not match Hash header
        expected: 132d96c356f5a5a17adb53b92716b73bb0b7a1581b85251458f32519a9a3a636
        header has: 00000cb2fd2996146d9d8f5d7863d2bc3d39d04beebb214112aa270196753afa
fail
```

▶ **mple 2: bad initial hash value**

Appended a 9 to the `Initial-hash:` header. This is a variation of Example 1. You should detect that the Initial-hash value does not match the hash of the message and not make assumptions about the length of the hash.

```
$ /pow-check headerfiles/abc.pow-20 testfiles/abc
ERROR: initial hashes don't match
       hash in header: 1010a7e761610980ac591359c871f724de150f23440ebb5959ac4c0724c91d919
       file hash: 1010a7e761610980ac591359c871f724de150f23440ebb5959ac4c0724c91d91
PASSED: leading bits is correct
PASSED: pow hash matches Hash header
fail
```

Example 3: bad Proof of Work value

Changing the `Proof-of-work:` header value (to `Yi!` in this example) will most likely result in a hash with a different number of leading bits than indicated in the header.

```
$ ./pow-check headerfiles/abc.pow-20 testfiles/abc
PASSED: initial file hashes match
ERROR: Leading-zero-bits value: 20, but hash has 0 leading zero bits
ERROR: pow hash does not match Hash header
        expected: b7c207d21d60e3778d49a8c46c2ee477f1d08153ca0c99c4b6295625c44db338
```

```
        header has: 00000cb2fd2996146d9d8f5d7863d2bc3d39d04beebb214112aa270196753afa
 fail
```

Example 4: bad count of leading zero bits

The Leading-zero-bits: header value was changed to 23 from 20.

`$ ./pow-check headerfiles/abc.pow-20 testfiles/abc`

`PASSED: initial file hashes match`

`ERROR: Leading-zero-bits value: 23, but hash has 20 leading zero bits`

`PASSED: pow hash matches Hash header`

`fail`

Example 5: missing header

You should check for presence and validity of the following headers:

- `Initial-hash:`
- `Proof-of-work:`
- `Leading-zero-bits:`
- `Hash:`

If any are missing, the program will fail the check. The `File`, `Iterations`, and `Compute-time` headers are just informative and don't need to be checked. Here's an example of a missing `Initial-hash:` header.

```
 ERROR: missing Initial-hash in header
 ▶  ED: leading bits is correct
    ED: pow hash matches Hash header
 fail
```

Your program should gracefully handle missing values in headers and may silently ignore extra header lines.

What to submit

Place your source code into a single zip file. If code needs to be compiled, please include a Makefile that will create the necessary executables.

We don't want to figure out how to run your program.

We expect to:

1. unzip your submission
2. run `make` if there's a `Makefile`
3. Set the mode of the programs to executable: `chmod u+x pow-create pow-check`
4. Run the commands as:

```
    ./pow-create #bits samplefile >sampleheader
    ./pow-check sampleheader samplefile
```

As with the previous assignment, if you are using python, you can submit either:

A. `pow-create` and `pow-check` scripts that run the program or

B. (preferably) programs named `pow-create` and `pow-check` that that start with a `#!` line followed by your code. For example:

```
#!/usr/bin/python3
print('Hello, world!') ...
```

If you are using java, you will have a makefile that compiles the class files and the `pow-create` and `pow-check` programs will be scripts that runs the appropriate java command with the arguments. The file `pow-create` will contain something like this:

```
#!/bin/bash
CLASSPATH=. java PowCreate "$@"
```

Testing

Test your scripts on an iLab machine to make sure they work prior to submitting.

Test that your programs gracefully handle any invalid input thrown at them. The programs should never ▶ or produce cryptic-looking stack traces regardless of what input you throw at it. Make sure that they you can validate the sample files, even if your pow-create program generates different proof-of-work values, as it almost certainly will. Make various modifications to the sample files to ensure that you detect errors correctly.