EECE 457 —- Hash assignment

Your assignment is to read this document through, and to type in the code examples in the gray boxes. There are four exercises at the end. If you have any difficulties, contact the instructor.

Getting started:

See if you have md5 or md5sum. MD5 ("Message Digest 5") is a common but outdated cryptographic hash function. Type the following at a command prompt:

```
bash-3.2$ echo foo | md5
d3b07384d113edec49eaa6238ad5ff00
bash-3.2$ echo -n foo | md5
acbd18db4cc2f85cedef654fccc4a4d8
```

A Mac should have an `md5` command; on some operating systems the command is `md5sum` instead of `md5`. If your local computer has neither, you can ssh to Bingsuns, which has both Tcl 8.5 and an `md5sum` command. If logged into Bingsuns, type this:

```
bingsuns2% echo –n foo | md5sum
acbd18db4cc2f85cedef654fccc4a4d8  –
```

Three minor issues you should note before we continue:

1. What's with that `–n` in "`echo –n`"? What does "echo" even mean?

   The Unix command "`echo stuff`" simply outputs the aforementioned stuff; in this case we pipe that stuff into the `md5` or `md5sum` command. The -n option outputs that stuff without a newline: "`echo –n foo`" outputs "`foo`" while "`echo foo`" outputs "`foo\n`". Note that these two outputs are distinct, and therefore have dramatically different hash values.

2. The MD5 hash output is 128 bits, displayed as a number with 32 hexadecimal digits.

3. Depending on the command you use, the output may or may not have a hyphen at the end. As you see above, the `md5sum` command on Bingsuns outputs a hash string followed by a hyphen. The hyphen means "I got this data from standard input," but nevermind what it means: what matters is that the output isn't a valid hexadecimal number if it contains these extra non-digit characters. This may break our code later, so watch out for that.

Whichever command you have, we will wrap it into a Tcl procedure as follows:

```
bash-3.2$ tclsh
% proc hash x { exec echo -n $x | md5 } ← Use md5sum if needed
% hash foo
acbd18db4cc2f85cedef654fccc4a4d8
```

The `[exec]` command invokes a Unix command and returns the output as a string. There are also native Tcl packages for computing hashes. Typing this *might* work:

```
% package require md5
2.0.7
md5::md5 -hex foo
ACBD18DB4CC2F85CEDEF654FCCC4A4D8
```

A package is a third-party library of commands that can be installed on your computer and dynamically loaded by your program. The most widely used package is Tk, the user-interface toolkit; but there are hundreds of others, including an `md5` package.

The problem is, you might not have it---you may see the error message “can't find package md5”. This is one of several reasons we avoid packages: they may or may not be installed on a given computer, while Unix commands are pretty universal. On the other hand, packages can be much faster than invoking a Unix command. If you didn’t get an error in the last box, try this:

```
% time {hash foo} 100
2514.8226 microseconds per iteration
% time {md5::md5 -hex foo} 100
49.67738 microseconds per iteration
```

Okay, wow: the native Tcl version is 50 times faster on my computer. If you have the `md5` package, you could redefine your [hash] procedure to get a 50-fold speedup.

In Tcl 8.5 we can also convert a hash from hexadecimal to decimal using `expr`:

```
% expr 0x[hash foo]
229609063533823256041787889330700985560
```

This works because Tcl follows the standard C/C++/Java/PHP/Perl/Etc convention that "0x" denotes a hexadecimal number. But unlike those other languages, Tcl natively handles arbitrarily large integers, like the 128-bit output of our `hash` procedure.

At this point you may get a cryptic error message, like "missing operand at _@_ in expression "0xabcdabcd -_@_". That means that the output of `[hash foo]` has some extra characters, like that hyphen we mentioned on the first page. This will confuse `[expr]`. If you have this problem, we can fix it with the `[lindex]` command, returning only the first part of the output, discarding extra spaces or hyphens or whatever:

```
% proc hash x { lindex [exec echo -n $x | md5] 0 } ← or md5sum
% hash foo
acbd18db4cc2f85cedef654fccc4a4d8
% expr 0x[hash foo]
229609063533823256041787889330700985560
```

The `[lindex]` command operates on lists ("lindex" is short for "list index,") but in Tcl, a list is nothing more than a bunch of words separated by whitespace. Therefore `"abcd -"` is a two-element list, and `[lindex "abcd -" 0]` will return "abcd".

Several applications of hash functions

We already discussed a few straightforward applications for hash functions:

1. **Tamper evidence:** if messages M_1 and M_2 are different, even very slightly, they will almost certainly have very different hashes. This allows us to publish a message M, and use its hash as a checksum to ensure that it hasn't been altered in transmission.

2. **Message surrogates**: with a digital signature algorithm, we can sign `[hash M]` instead of M itself. Signing a few dozen bytes is far easier than signing a 10MB file.

   Notice the subtle difference in meaning: instead of "I hereby agree to message M," the signature of a hash means, "I hereby agree to some message that hashes to H." This means that someone who can find hash collisions could trick us into signing one contract M_1 and then holding us to another contract M_2, if hash(M_1)=hash(M_2). Only if our hash function is secure can we use `[hash M]` as a surrogate for M.

3. **Commitments**: we can keep a message secret, but publish a hash of it to commit to its value. We used this to "flip a coin" over the Internet, but we can use the same technique to achieve many useful and interesting effects.

   Note that (2) and (3) make it possible to sign a file that you can't read, if someone keeps a message secret and sends you its hash to sign as a commitment. Why you would ever want to sign something you can't read is a matter for a future lecture.

More applications: timestamping with hash chains

Suppose you want to publish an important piece of information online, and be able to prove that it was published by a certain time. For example, you want to publish your screenplay for *Assembly Language: the Movie*, and be able to sue someone who misappropriates your work, demonstrating that your publication date precedes theirs.

You can't simply publish your message with a date, because dates can be faked, along with file creation times. You could mail the message to yourself or have it published, and then rely upon the courts to trust the publisher or mail service; but suppose we want a way to do this without requiring any trusted third party.

One approach is creating a hash chain. It works like this: a group of people publish their work in an online forum, and each person begins his or her work with a hash of someone else's work chosen from the forum.

```
% proc chainlink {x h} {
     return "[hash $x] (time==[clock seconds])\n$h"
}
% set foo [chainlink start "Hi from Alice"]
EA2B2676C28C0DB26D39331A336C6B92 (time==1392310645)
Hi from Alice
% set bar [chainlink $foo "Message from Bob"]
CB97F55B1A2108442555CC45CB0C50E0 (time==1392310654)
Message from Bob
```

Assuming the hash is trustworthy, Bob's message could not have been written without knowledge of Alice's message; if Bob's message is published, it serves as evidence that Alice's message was published previously, or at least known to Bob previously. It also guarantees that Alice's message is unaltered, for an alteration would change the hash.

Over time, a long string of users chaining each other's messages will produce a record that serves as evidence of each message's authenticity and time of publication. Note that the dates in a message can be bogus, but a chain of 1000 messages in a web forum can provide strong evidence of a message's provenance.

What if an attacker can tamper with a database of message forum comments going back 10 years? In order to fabricate a message backdated to 2003, you could invent a long hash chain of messages from other people, inventing a 10-year-long conversation which would then have to be injected into a forum. Is that possible? Maybe, but we can prevent it with the next application.

Hashcash

A *proof-of-work* function is a function that requires a predictable amount of effort to compute. For example, suppose I give you a random string X, challenge you to find any other string S such that [hash XS] ends with a “001”. Type this:

```
% proc tailhash {x n} {
     string range [hash $x] end-$n end
}
% tailhash foo 2     ;# the last 3 digits of [hash foo]
4d8
% tailhash foo1 2
8bb
```

We can now find an input string that starts with “foo,” and whose hash has the suffix “001,” by looping a counter indefinitely until we get a match:

```
% set i 0; while {[tailhash foo$i 2] ne "001"} {incr i}
% set i
1364
% hash foo1364
8340d435bd9f3c9c4e7ace8f9d2a3001    ←it works!
```

Let’s wrap this into a procedure, that effectively solves our challenge problem:

```
% proc hashcash {prefix suffix} {
     set n [string length $suffix]; incr n -1
     while {[tailhash $prefix[incr i] $n] ne $suffix} {}
     set i
}
% hashcash foo 001   ;# find S such that [hash foo$S] ends 001
1364
```

Let’s time this:

```
% time {hashcash foo 001}
3420696 microseconds per iteration
% time {hashcash foo 0001}            (surf the web for a while)
48620457 microseconds per iteration
```

If a hash function is secure, then the fastest way to find a hash with a given suffix is by brute-force, with a number of iterations that grows exponentially with the length of the suffix. To be specific, the probability of a random hash ending with hex digits “001” is 16^{-3} = 1/4096; any specific d-digit suffix occurs with probability 16^{-d}. We can show that the expected number of guesses needed to find a specific suffix is the reciprocal of its probability---so on average, it will take 16^{5} tries before you find a hash ending “12345”.

Let’s demonstrate this by trying to find a two-digit suffix “ab”, which should occur with probability 1/256. Note that the output of `[hashcash]` is not only the desired suffix, but also the number of loop iterations needed to find it. We can rut it 100 times with different prefixes, and average its return value, which should be around 256:

```
% for {set i 0; set out ""} {$i<100} {incr i} {
     lappend out [hashcash $i--- ab]
     puts $out
}
553
553 190
553 190 136        [growing list of 100 return values]
...
% hash 0---553
b59d33740e7d9ef3a51a93fddf9e2eab    ←ends in ab:  it works!
% hash 1---190
7202574b8c4e95b7a9c242233e0b65ab    ←ends in ab:  it works!
% expr [join $out +]     ;# sum of all 100 return values
23591
```

Seems legit. The `[hashcash x ab]` command returned an average of 235.91 iterations. If you can speed up your `[hash]` command with the md5 package or otherwise have a lot of time to kill, you can verify the same result for a longer suffix.

We can now give someone a computational challenge engineered to take a certain amount of time. If it takes 2.5ms per hash trial, then finding a 6-digit suffix takes 11.65 hours. On top of that, it takes only a single hash for you to verify the answer once it is found. This allows you or anyone else to verify that someone has done the work.

This concept was originally proposed by Adam Back, who called it *hashcash*. Hashcash problems challenge you to find inputs whose hash, written in binary, end in a certain number of zeroes. An input that produces a suffix of m zeroes is called an *m-bit hashcash,* that should require 2^{m} trials to find by brute force.

But why do we care?

What can you do with a proof-of-work function? Here are a few applications:

1. **Preventing forgery of hash chains**. In the previous section, we imagined a web forum where each post included a hash of a previous post. Instead, each user can include a hashcash of a previous post:

   ```
   % proc chainlink {x h} {
         set hash [hash $x]
          return "$hash ([hashcash $hash 001]) ([clock seconds])\n$h"
   }
   % set foo [chainlink start "Hi from Alice"]
   ea2b2676c28c0db26d39331a336c6b92 (4498) (1392319144)
   Hi from Alice
   ```

   This requires a specified amount of work to be performed in order to create each new post, so that a very long chain would take too much time to forge.

2. **Digital currency.** Since hashes with specific suffixes take time to find, and large suffixes are particularly rare, you can imagine them as valuable tokens that have to be “mined.” Hashcash is called “hashcash” because it is somewhat analogous to money, at least in the sense that it is an easily exchanged product of labor.

   The Bitcoin virtual currency is based on a proof-of-work function similar to hashcash to grow a hash chain of financial transaction messages. Participants receive financial rewards for computing successful entries in the hash chain.

3. **Spam prevention**. A user can, when sending an email, spend a few seconds of work to find a hashcash based on the sender’s and recipient’s email address. This can be quickly verified by the recipient, showing that the sender performed a small amount of work to send the email. In theory, an email client could refuse to accept emails that don’t have a hashcash header, and a spammer would have to spend millions of seconds to send millions of emails.

   In practice, however, this application has not worked, due to the practical issues in email services, and attacks available to spammers.

4. **Proof of computing power**. If you want to join a distributed computing project, a hashcash problem can be used to measure and verify the level of computing power you have available.

Exercises:

1. **Pseudo-random number generation:** Using your `hash` procedure, write a procedure `[PRNG seed length]`, that uses a seed to output $length bytes, in hex:

```
% PRNG secretseed 30
3af854ec902e7706004df5027e5fdb099ab1e30c2a540c23b11d39afa07c
```

The seed should be used to generate the sequence, with a specific seed generating the same output each time. Someone without knowledge of the seed should not be able to generate or predict the output, even given previous output to analyze.

Look up the `string` command in the Tcl documention at www.tcl.tk --- this has many useful subcommands for manipulating strings and substrings. You can build up a string using the `append` command, or the corresponding `lappend` command for lists.

NOTE: for this assignment, please follow these guidelines.

a. Your procedure should *return* a value, and not *print* a value. Printing output to the screen is not the same as providing a return value.

Likewise, the `seed` and `length` are arguments to the procedure, and should not be input prompted from the user.

b. Your procedure should work with any number, not just 30 or other small numbers.

c. Your procedure should output the requisite number of *bytes*, not bits or whatever else.

d. You can just hand in your source code, on paper. You don't need to submit it in a format that I can run.

2. Find a string that starts with your B-number, whose md5 hash ends with as many zero bits as you can find.

3. How many passwords are possible in each of the following cases? Write each answer as a number, and write its log base 2:

a. My password is a string of 10 lowercase letters.
b. My password is 5 characters that can be upper- or lowercase letters or digits.
c. My password is my birthday, written MM/DD/YY.
d. My password is four English words drawn from a novel (of your choice)