15-122: Principles of Imperative Computation

Lab Week 7

Tom Cortina, Rob Simmons

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we’ve learned, look at problems from a different perspective and allow you to ask questions about topics you don’t understand. Feel free to talk with your neighbors about the problems!

Grading: You should finish (1.a), (1.b), and (1.c) for partial credit. You should complete at least one additional problem for full credit: we recommend starting with (1.d), as it is the most straightforward one.

Finding collisions in hash functions

Recall that a hash function \( h(k) \) takes a key \( k \) as its argument and returns some integer, a hash value; we can then calculate \( \text{abs}(h(k) \% m) \) to get an index into our hash table. In this lab you will be using various hash functions on strings and examining possibilities for collisions.

Let string \( s \) of length \( n \) \( (n > 0) \) be denoted as \( s_0 s_1 s_2 ... s_{n−2} s_{n−1} \), where \( s_i \) is the ASCII value of character \( i \) in string \( s \). (A partial ASCII table is given to the right.) We define five hash functions as follows:

\[
\text{hash}_\text{len}: h(s) = n
\]

\[
\text{hash}_\text{add}: h(s) = s_0 + s_1 + s_2 + ... + s_{n−2} + s_{n−1}
\]

\[
\text{hash}_\text{mul32}:
\]

\[
h(s) = \ldots ((s_0 \cdot 32 + s_1) \cdot 32 + s_2) \cdot 32 + ... + s_{n−2} \cdot 32 + s_{n−1}
\]

\[
\text{hash}_\text{mul31}:
\]

\[
h(s) = \ldots ((s_0 \cdot 31 + s_1) \cdot 31 + s_2) \cdot 31 + ... + s_{n−2} \cdot 31 + s_{n−1}
\]

\[
\text{hash}_\text{lcg}:
\]

\[
h(s) = f(f(...f(f(s_0) + s_1) + s_2) + ... + s_{n−2}) + s_{n−1}
\]

where \( f(x) = 1664525 \cdot x + 1013904223 \)

These five hash functions have been implemented for you and can be run from the command line:

\`
% hash_len
Enter a string to hash: bar
hash value = 3
hashes to index 3 in a table of size 1024
Another? (empty line quits): snafu
hash value = 5
hashes to index 5 in a table of size 1024
Another? (empty line quits):
\`

Note that the command line hashing tool also reports where the element with the given key will hash given a table size of 1024.
The first two exercises require you to mathematically reverse-engineer the simpler hash functions:

(1.a) Find three or more strings, each string containing three or more characters, that would always collide because they have the same hash value using hash_add.

(1.b) Find three or more strings, each string containing three or more characters, that would always collide because they have the same hash value using hash_mul32. (Hint: use the fact that 32 is a power of 2.)

The hash_mul31 function is slightly more complicated – it’s actually the default string hashing function used in Java! It’s still possible to find hash value collisions by doing some math with pen and paper, though.

(1.c) Find three or more strings, each string containing three or more characters, that would always collide because they have the same hash value using hash_mul31.

The more complicated a hash function gets, the more you may need to rely on “brute force search” – trying a lot of words and seeing which ones match.

(1.d) Implement your own version of hash_mul31 as a function that takes a single non-empty string as its argument and returns an integer representing the hash value for that string using the formula given on the previous page. Demonstrate that it works correctly by comparing the results of this function in coin with your answers from Exercise 3. Your function does not need to compute the hash index for a table of size 1024.

It should be very easy to cut-and-paste-and-modify this function to create your own implementation of hash_lcg as well.

(1.e) Using hash_mul31, find three or more strings, each containing three or more characters, that do not have the same exact hash values but do collide in a hash table of size 1024.

(1.f) Using hash_lcg, find three or more strings, each containing three or more characters, that do not have the same exact hash values but do collide in a hash table of size 1024.

(1.g) Only two words in the Scrabble dictionary have the same hash value under hash_lcg: “charmeuse” and “historicizes”. (The hash value is 706668240.) Can you find two other strings with the same hash value?

(1.h) The empty string has the hash value 0 under hash_lcg, and the closest any Scrabble dictionary word comes to this hash value is “gristlier” (the hash value is -17760). Can you find a non-empty string with a hash value closer to 0?

Our hash_lcg produces a hash value that is a 32-bit C0 integer. More complex hash functions produce more bits: “SHA256” is a hashing algorithm that produces a 256-bit hash value, and “Skein 1024 1024” is a hash function that produces a 1024-bit hash value.

For a class of hash functions called cryptographic hash functions, brute force search is thought to be the best known way to create collisions. If you’ve ever heard of “mining Bitcoins,” it largely involves using brute-force search to solve problems like (1.h) for SHA256.