15-122: Principles of Imperative Computation

Lab 7: #pun — Shyam Raghavan, 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. We encourage discussing problems with your neighbors as you work through this lab!

Grading: You should finish (1.a), (1.b), and (1.c) for two points. You should complete any two additional problems for three points: we recommend starting with (1.d).

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 $\mathrm{abs}(h(k)\%m)$ to get an index into our hash table. In this lab you will be examining various hash functions and exploiting their inefficiencies to make them collide.

Let string s of length n $(n > 0)$ be denoted as $s_0 s_1 s_2 \ldots s_{n-2} s_{n-1}$, 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:

`hash_len`: $h(s) = n$

`hash_add`: $h(s) = s_0 + s_1 + s_2 + \cdots + s_{n-2} + s_{n-1}$

`hash_mul32`:

$$h(s) = (\ldots((s_0 * 32 + s_1) * 32 + s_2) * 32 \cdots + s_{n-2}) * 32 + s_{n-1}$$

`hash_mul31`:

$$h(s) = (\ldots((s_0 * 31 + s_1) * 31 + s_2) * 31 \cdots + s_{n-2}) * 31 + s_{n-1}$$

`hash_lcg`:

$$h(s) = f(f(\ldots f(f(f(s_0) + s_1) + s_2) \cdots + s_{n-2}) + s_{n-1})$$

where $f(x) = 1664525 * x + 1013904223$

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):
```

Partial ASCII Table

32	20	␣	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_			

Note that the command line hashing tool also reports where the element with the given key will hash given a table size of 1024. We care, however, about hash functions that always collide under a table of any size. Thus, you'll be looking for hash functions that hash to the same value.

The first exercise requires you to mathematically reverse-engineer one of 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`.

Now, you'll implement one of the hash functions and then use your implementation to find collisions that always occur because the strings hash to the same value. 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.b) 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 the answers from the `hash_mul31` binary. 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.c) Using your implementation of `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.

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) 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.e) 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.f) 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? If you find any that are closer than "gristlier", submit it to the course infrastructure (ask TA your about this) and you'll be added to the scoreboard!

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" 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.f)** for SHA256.