Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

CSI2110 (Winter 2021)

Programming Assignment 2 (15%)

Prof. WonSook Lee

Note: You are not allowed to post any content in public.

This programming assignment is composed of three tasks, which are not related to each other.

TASK 1 (5%): A recursive hash function

(a) (3%) Create a recursive hash function where input is words and output is numerical values. The hash table size is **2^M**. The function is defined recursively as follows:
  (i) fn(“”) = 0 where “” = empty word
  (ii) fn(previousWord + letter) =((fn(previousWord) x 33) XOR ASCII(letter)) mod 2^M where XOR is for the bitwise operator. **0110** XOR **1010** = **1100** and ASCII(letter) is for ASCII code. e.g. ASCII(“A”) = 65, ASCII(“b”) = 98, ASCII(“m”) = 109

As an example for M = 10, we can find the value of “Abm” recursively as follows:

- fn(“”) = 0
- fn(“A”) = fn(“”) x 33 XOR ASCII(“A”) mod 2^M = 0 XOR 65 mod 2^{10} = **0000000** XOR **01000001** mod 2^{10} = 65
- fn(“Ab”) = fn(“A”) x 33 XOR ASCII(“b”) mod 2^M = 65 x 33 XOR 98 mod 2^{10} = 2145 XOR 98 mod 2^{10} = 3
- fn(“Abm”) = fn(“Ab”) x 33 XOR ASCII(“m”) mod 2^M = 3 x 33 XOR 109 mod 2^{10} = 99 XOR 109 mod 2^{10} = 14 mod 1024 = 14

Your program will need to have input and output as follows:

- Input : M (return) and a word (return)
  - e.g.
    10
    Abm
- Output : an integer of **2^M** bits
  - e.g
    14

(b) (1.5%) Write a program to calculate how many words with a given length N there are with a given hash value K and what these words are. N will be given between 1 and 5 inclusively, K between 0 and 2^M inclusively and M between 0 and 25 inclusively. For simplicity, consider the input words consisting of only lowercase letters of the English alphabet. It is like calculating how many collisions there are with the same hash value.

Your program will need to have input and output as follows:

- Input : N K M (return)
  - e.g. (There are words with length 3 whose hash value is 10 in a hash table 2^12. FInd how many words and what they are.)

    3 10 12
- Output : an integer and words
  - e.g (There are 3 words with length 3. They are below)

    3

    hsq pcy xsa

(c) (0.5%) Write big-Oh in terms of input words with a given length N, and size of hash table, K, in the tasks a and b big-Oh. and your analysis of them.

<u>TASK 2 (2%)</u>: Bit folding for text

Create a hash function where input is a paragraph and output is numerical values. It hashes an ascii string using bit folding as follows:

For example, we hash the string ‘lorem ipsum dolor’ into a 32 bit integer.

* ascii characters are 1 byte each (that’s 8 bits)

* integers are 32 bits

So the letters “lore”:

l — 01101100

o — 01101111

r — 01110010

e — 01100101

“folding” our bytes together (right to left) we get:
01100101 01110010 01101111 01101100
This translates to a integer value of 1701998444

The next four characters will be “m ip”, which yield the value:
01110000 01101001 00100000 01101101
This translates to a integer value of 1885937773

Then taking XOR these we get: 00010101 00011011 01001111 00000001 → 354111233

Then we continue to read four characters. If the last remaining characters are less than 4, first use 10000000 and then if needed, use one or more times 00000000. This is called as padding.

Your program will need to have input and output as follows:

- Input : text (return)
  - e.g. data structure is fun
- Output : an integer of 32 bits in the decimal representation
  - e.g 2050110814

<u>TASK 3 (8%)</u>: SHA-320 is one of cryptographic hash functions which are used to generate passwords. A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function which is practically infeasible to invert.

Useful Links:

- You can test several cryptographic hash functions - https://www.mobilefish.com/services/hash_generator/hash_generator.php
- Wikipedia page for SHA-2 - https://en.wikipedia.org/wiki/SHA-2
- Youtube video for SHA - https://www.youtube.com/watch?v=DMtFhACPnTY)

The overall idea for the SHA-320 is to break a given input message into 1024-bit chunks and then process one by one to produce one final hash value.

1. hash values h0, h1, h2, … h7 are initialized
2. array of round constants k[0], k[1], … k[63] are initialized
3. Pre-processing (Padding) to make the given text to be desired length (`length a multiple of 512 bits)` by adding 1000...00 in the end
4. For each 512-bit chunks, update h0, h1, h2, … h7 with intermediate parameters a, b, c, … , h and k[0], .. [63] then produce a hash value for the given input message.

Our Pseudocode for the SHA-256 algorithm follows. By modifying SHA-256, you will produce SHA-320.

```
Note 1: All variables are 32 bit unsigned integers and addition is calculated
modulo 2^32
Note 2: For each round, there is one round constant k[i] and one entry in the
```

```
message schedule array w[i], 0 ≤ i ≤ 63
Note 3: The compression function uses 8 working variables, a through h
Note 4: Big-endian convention is used when expressing the constants in this
pseudocode,
    and when parsing message block data from bytes to words, for example,
    the first word of the input message "abc" after padding is 0x61626380

Initialize hash values:
(first 32 bits of the fractional parts of the square roots of the first 8
primes 2..19):

h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Initialize array of round constants:
(first 32 bits of the fractional parts of the cube roots of the first 64
primes 2..311):
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
0x923f82a4, 0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
0x9bdc06a7, 0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
0x5cb0a9dc, 0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
0x06ca6351, 0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
0x81c2c92e, 0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
0xf40e3585, 0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
0xbef9a3f7, 0xc67178f2

Pre-processing (Padding):
begin with the original message of length L bits
append a single '1' bit
append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K + 64
is a multiple of 512
append L as a 64-bit big-endian integer, making the total post-processed
length a multiple of 512 bits
such that the bits in the message are L 1 00..<K 0's>..00 <L as 64 bit
integer> = k*512 total bits

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
```

```
    create a 64-entry message schedule array w[0..63] of 32-bit words
    (The initial values in w[0..63] don't matter, so many implementations zero
them here)
    copy chunk into first 16 words w[0..15] of the message schedule array

    Extend the first 16 words into the remaining 48 words w[16..63] of the
message schedule array:
    for i from 16 to 63
        s0 := (w[i-15] rightrotate  7) xor (w[i-15] rightrotate 18) xor (w[i-
15] rightshift  3)
        s1 := (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i-
2] rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] + s1

    Initialize working variables to current hash value:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7

    Compression function main loop:
    for i from 0 to 63
        S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
        ch := (e and f) xor ((not e) and g)
        temp1 := h + S1 + ch + k[i] + w[i]
        S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
        maj := (a and b) xor (a and c) xor (b and c)
        temp2 := S0 + maj

        h := g
        g := f
        f := e
        e := d + temp1
        d := c
        c := b
        b := a
        a := temp1 + temp2

    Add the compressed chunk to the current hash value:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append
```

```
h6 append h7
```

Produce SHA-320 algorithm, which is identical in structure to SHA-256, but:

- the message is broken into 1024-bit chunks,
- the initial hash values and round constants are extended to 64 bits,
- there are 80 rounds instead of 64,
- the message schedule array w has 80 64-bit words instead of 64 32-bit words,
- to extend the message schedule array w, the loop is from 16 to 79 instead of from 16 to 63,
- the round constants are based on the first 80 primes 2..409,
- the word size used for calculations is 64 bits long,
- the appended length of the message (before pre-processing), in *bits*, is a 128-bit big-endian integer, and
- the shift and rotate amounts used are different.
- the output is constructed by omitting `h5, h6` and `h7`.

```
SHA-320 initial hash values (in big-endian):

h[0..7] := 0xcbbb9d5dc1059ed8, 0x629a292a367cd507, 0x9159015a3070dd17,
0x152fecd8f70e5939,
           0x67332667ffc00b31, 0x8eb44a8768581511, 0xdb0c2e0d64f98fa7,
0x47b5481dbefa4fa4

SHA-320 round constants:

k[0..79] := [ 0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f,
0xe9b5dba58189dbbc, 0x3956c25bf348b538,
              0x59f111f1b605d019, 0x923f82a4af194f9b, 0xab1c5ed5da6d8118,
0xd807aa98a3030242, 0x12835b0145706fbe,
              0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2, 0x72be5d74f27b896f,
0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
              0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3,
0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,
              0x2de92c6f592b0275, 0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4,
0x76f988da831153b5, 0x983e5152ee66dfab,
              0xa831c66d2db43210, 0xb00327c898fb213f, 0xbf597fc7beef0ee4,
0xc6e00bf33da88fc2, 0xd5a79147930aa725,
              0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc,
0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed,
              0x53380d139d95b3df, 0x650a73548baf63de, 0x766a0abb3c77b2a8,
0x81c2c92e47edaee6, 0x92722c851482353b,
              0xa2bfe8a14cf10364, 0xa81a664bbc423001, 0xc24b8b70d0f89791,
0xc76c51a30654be30, 0xd192e819d6ef5218,
              0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8,
0x19a4c116b8d2d0c8, 0x1e376c085141ab53,
              0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63,
0x4ed8aa4ae3418acb, 0x5b9cca4f7763e373,
              0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc, 0x78a5636f43172f60,
0x84c87814a1f0ab72, 0x8cc702081a6439ec,
              0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915,
0xc67178f2e372532b, 0xca273eceea26619c,
```

```
                0xd186b8c721c0c207, 0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178,
0x06f067aa72176fba, 0x0a637dc5a2c898a6,
                0x113f9804bef90dae, 0x1b710b35131c471b, 0x28db77f523047d84,
0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
                0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a,
0x5fcb6fab3ad6faec, 0x6c44198c4a475817]

SHA-320 Sum & Sigma:

S0 := (a rightrotate 28) xor (a rightrotate 34) xor (a rightrotate 39)
S1 := (e rightrotate 14) xor (e rightrotate 18) xor (e rightrotate 41)

s0 := (w[i-15] rightrotate 1) xor (w[i-15] rightrotate 8) xor (w[i-15]
rightshift 7)
s1 := (w[i-2] rightrotate 19) xor (w[i-2] rightrotate 61) xor (w[i-2]
rightshift 6)
```

Your program will need to have input and output as follows:

- Input : text (return)
  - e.g. The COVID-19 pandemic has changed everything, from how we work, to how we shop, from what and how we eat to how we interact socially and what we care about politically.
- Output : 320 bits in Hexadecimal format
  - e.g A466FBF66423B868CAC92CD96D2A5C142417B1076B7E2AFE180116A87175915F5D40589AA73AAB02
  - This hex number corresponds to 320 bits 1010010001100110111111011111101100110010000100011101110000110100011001010110010010010110011011001011011010010101001011100000101000010010000010111101100010000011101101011011111100010101011111110000110000000000010001011010101000011100010111010110010001010111110101110101000000010110001001101010100111001110101010101100000010

Marking Schene

Correctness of solutions provided: 50%
Quality of programming: 50%

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]