Information Security Coursework1

Deadline 10/11/2019

1.1

1	0	1	1	1	0	1	
2^6 x1	2^5 x0	2^4 x1	2^3 x1	2^2 x1	2^1 x0	2^0x1	
64+	0+	16+	8+	4+	0+	1	$=93_{10}$

1.2

Given the rule of ¬(A ^ B) = (¬A) ᵛ (¬B), this (the expression given in the question) reduces first to ¬(¬(A ^ B)) and then, given that ¬(¬A) = A, we're left simply with A ^ B. Therefore, the answer is:

0	1	0	1	0	1	0	1
1	0	1	0	1	1	1	1
0	0	0	0	0	1	0	1

2.1

Answer: 15 (Explanation: 15/15=1 and 120/15=8)

Answer: 120 (Explanation: 120/120=1

Answer: 12 (Explanation: 72/12=6 and 84/12=7)

2.2

Answer: 2x2x2x2x3x3

Explanation:

144 is 12x12; and 12 is 2x2x3

2.3

Answer: It equals to 0 mod 3 = 0.

Explanation:

Following the rule of modular multiplication, which says that (A * B) mod C = (A mod C * B mod C) mod C. We can perform first a mod on A and see that this will evaluate to x = 0, because A is exactly divisible by 3 (45, the last two digits of a are dividable by 3). Then looking at the first row in the multiplication table provided, we know that whatever B mod 3 will evaluate to, say y, after multiplying y with x, it will still be 0. And so, then performing a modulo 3 on 0 evaluates to 0 again.

3

Plain text	i	n	f	o	r	m	a	t	i	o	n	s	e	c	u	r	i	t	y
key	r	u	n	r	u	n	r	u	n	r	u	n	r	u	n	r	u	n	r
Key Index	17	20	13	17	20	13	17	20	13	17	20	13	17	20	13	17	20	13	17
+ letter Index	8	13	5	14	17	12	0	19	8	14	13	18	4	2	20	17	8	19	24
=	**25**	**33**	**18**	**31**	**37**	**25**	**17**	**39**	**21**	**31**	**33**	**31**	**21**	**22**	**33**	**34**	**28**	**32**	**41**
mod 26 =	25	7	18	5	11	25	17	13	21	5	7	5	21	22	7	8	2	6	15
cipher	Z	H	S	F	L	Z	R	N	V	F	H	F	V	W	H	I	C	G	P

4.1

inoonfarmotiensrcuyit

4.2

(assuming the letters in word are multiples of 3)

word = the word we're deciphering

for l_i where l_i means letter at position i, begin with

i =1

Starting from left to right in *word*,

While i <*word*.length

1. Copy l_i to a temporary variable *temp*
2. Copy l_{i+1} to position i, overriding what's there
3. Copy l_{i+2} to position i +1, overriding what's there
4. Copy l from *temp* to position i +2, overriding what's there

5. $i = i + 3$

endWhile

Demonstration:

To shorten my demonstration, assume we first divide inoonfarmotiensrcuyit into sections of 3 -> ino onf arm oti ens rcu yit

If the algorithm is then applied to each of the sections,

At line 2 in the algorithm above, the sections would look like this:

$$\text{nno}_{temp=i} \quad \text{nnf}_{temp=o} \quad \text{rrm}_{temp=a} \quad \text{tti}_{temp=o} \quad \text{nns}_{temp=e} \quad \text{ccu}_{temp=r} \quad \text{iit}_{temp=y}$$

At line 3, like this:

$$\text{noo}_{temp=i} \quad \text{nff}_{temp=o} \quad \text{rmm}_{temp=a} \quad \text{tii}_{temp=o} \quad \text{nss}_{temp=e} \quad \text{cuu}_{temp=r} \quad \text{itt}_{temp=y}$$

And at line 4, like this:

$$\text{noi} \quad \text{nfo} \quad \text{rma} \quad \text{tio} \quad \text{nse} \quad \text{cur} \quad \text{ity}$$

(In this demonstration, all that is left to do, is to concatenate the sections to get:

$$\text{noinformationsecurity}$$

4.3

Let **d** be the number for the shift instruction (that is how many positions an element should be shifted), **n** be the length of pt and the gcd for **n** and **d** be 1, then the description of the shifting algorithm is the following:

- Start from the last index
- Copy content to the index indicated by the right shift instruction, counting to the right and wrapping round the array if needed (in this case, the shift instruction is 2)
    - *Note, at the first copy operation, you'll have to first copy the content of the destination to a temp variable (in this case it's the integer 2) before overriding it.
- Have a counter to record each copy/paste action
- Designate the new destination, to be the previous source and the new source to be [the content of the index of] the previous destination + 1
- Repeat this until you get to your **n**-1 copy/past action, in which case, at your **n**th copy, instead of your new source to be the previous destination +1, your source is the stored temp variable.
- After n copy/paste actions, the shift operation is completed and the entire pt is permutated.

Eg (of course, instead of numbers they could be letters):

{1, 2, 3, 4, 5}//start from the last index (blue for source, red for dest).

temp=2

{1, 5, 3, 4, 5} //underlining signifies an element which is in the right place already

{1, 5, 3, 4, 3}

{1, 5, 1, 4, 3}

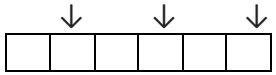
{4, 5, 1, 4, 3}

{4, 5, 1, 2, 3} //pasted in from temp

Let **d** be the number for the shift instruction (that is how many positions an element should be shifted), **n** be the length of pt and the gcd for **n** and **d** be such that it is equal to n/m, then the description of the shifting algorithm is the following:

- View the array as divided into n/m sections

- Since the sections are of the same length, have a pointer that points at the same index number for each group, starting of from the most left index in each set. Eg:



*Note, can be implemented with loops and temp variable, with the new source being calculated each time as the pointer n (pointing to the destination where to copy the values to) - d, n-d*2, n-d*3…etc.

- Copy content from each pointed index, as many places to the right, as indicated by the right shift instruction (wrapping round the array if needed - in this case, the shift instruction is 2)
    - *Note, at the beginning of each round, you'll have to first store the content of the destination to where the first element is overridden into a temp variable (in this case it's the integer 2) before overriding it, so that you can later replace it (at the other end of the array).
- Have a counter to record each copy/paste round/set
- Designate the new destination, to be the previous source and the new source to be [the content of the index of] of the next index to the left within each section
- Repeat the rounds until you've reached the left most index of the sections
- Assume there are j elements in a section, after j rounds of copy/paste sets, the shift operation is completed and the entire pt is permutated. (Note, arbitrary random letters can always be appended so that the pt can always be divided in more than one section.)

Eg (of course, instead of numbers they could be letters):

{1, 2, 3, 4, 5, 6} //start from the right most index.

temp = 6

{1, 6, 3, 2, 5, 4}

{1, 6, 3, 2, 5, 4}

temp = 5

{5, 6, 1, 2, 3, 4}

5.1

Encoding:

Let the binary word be halved (for simplicity assume equal halves)

Name the left halve L0 and the right half R0.

Apply the above algorithm f to R0 with the key being the fact that we're doing a 'forward shift' by one to each block of 3 (as specified in 4.2) and assign the results to y: f(R0, key) =y

Now xor y with L0 to get R1.

Put (the unchanged) R0 to the left to become L1 (for the next round).

Put the computed R1 to the right of L1 and concatenate the two, forming R1 + L1.

Now start again another round (The number of rounds is arbitrary although the same for a given method. Also, keys can change from round to round but would be related to each other.)

Decoding:

Note, that since L1 is in fact R0, we can compute y by following the same function as in the encoding stage.

Then, to find L0 (or the left part of any given previous round number n) we xor R1 with y to obtain L0. This is owed to the fact that xoring a binary string z with x, when z itself was obtained by xoring x and y, gives you binary string s = y.

(Explanation: in the case of (a column in z being) a 0:

If it's because both x and y had 1s, then z xor x also = 0.

If it's because both x and y had 1s, then z will have been computed to 0 and so doing xor again, now with z and x, will come to 1.

In the case of (a column in z being) a 1:

If the reason for that is because y had the 1, then by definition, x must've had the 0 and so now xoring z with x, 1 with 0, produces a 1, that is what y had.

If the reason for that is because y had the 0, then x must've had the 1, and so now xoring 1 with 1 produces a 0, that is indeed what y had.)

Now knowing L0, we know that it is in fact R-1 and so on… reversing rounds until we get back to the original binary word we started with.

5.2

0	1	1	1	1	0	1	0	0	0	0	1	PT
						0	1	0	1	0	0	y
						0	1	1	1	1	0	L0
						0	0	1	0	1	0	R1 (L0 xor y)
1	0	0	0	0	1	0	0	1	0	1	0	L1 (R0) + R1