Problems for Op 2018

Friday, 16 November 2018
Copyright © 2018 MSOE

- Unless explicitly stated by a problem, you may assume that all user inputs are valid.
- Unless explicitly stated by a problem, the precise format or wording of your outputs is not critical as long as they are completely clear.

Problem 1: Contraction Expander – 10 Points

Write a program that will prompt the user to enter a sentence, then search for and expand the following I contractions – I'm → I am, I've → I have, and I'll → I will. Do not change I'd or any other contractions.

Sample session (bold is program-generated output):

```
In [1]: run prob1.py

Enter sentence: I'd like a candy bar but I'll settle for a carrot.
I'd like a candy bar but I will settle for a carrot.
```

Problem 2: I Before E Except After C – 10 Points

Write a program that will accept a sentence entered by the user and then report all words that violate the rule "I before E except after C" by displaying the position of the offending word by word count and the word itself. Interpret the rule literally – you do not need to accommodate any of the exceptions to this rule, nor are you correcting spelling.

Sample session (bold is program-generated output):

```
In [2]: run prob2.py

Enter sentence: Either it is science or just weird that I received
sufficient caffeine
1 :  Either
7 :  weird
12 :  caffeine
```

Problem 3: Clockface Coordinates – 10 Points

Consider an analog clock face where the center is at Cartesian coordinates 0,0, 12 o'clock is on the positive y axis, and the minute hand is of unit length. Prompt the user for the number of minutes past the hour, and then display the Cartesian x and y coordinates of the tip of the minute hand rounded to two decimal places. You can assume the user enters the minutes as a single integer from 0 to 59. Do not worry about potential -0 outputs.

Sample session (bold is program-generated output):

```
In [3]: run prob3.py

Enter minutes: 45
Minute hand is at (x,y): -1.00, -0.00
```

Problem 4: Coin Toss Simulator – 20 Points

You and a friend each have ten dollars and have decided to wager one dollar per fair coin flip with the winner of the flip receiving one dollar from the loser. You continue to wager one dollar and flip until one of you is out of money. Write a program that will simulate this game. Your program should prompt the user for how many simulations to run, and upon completing the simulations, will display to the console the number of times you won all the money, the number of times you lost all your money, and the average number of flips it took to either win or lose.

Sample session (bold is program-generated output):

```
In [6]: run prob4.py

Enter number of simulations: 10
Wins: 4  Loses: 6  Avg flips to win/loss:  151.2
```

Problem 5: RLE ASCII Art Decompressor – 20 Points

In an effort to help reduce the astonishing amount of bandwidth used to transmit ASCII art over the Internet, RLE compression has been adopted to shrink the file size. This technique remains an ASCII format, thus has some limitations, but is still human readable.

The [text] file format is as follows:

<length><char><length><char>…<eol>
<length><char><length><char>…<eol>
…<eol>

Where length is a single digit numeral 1-9, and char is the character to be repeated length times. Since the length field is only one digit, a run of more than nine of the same character will require multiple length/character instances. Char can be any printable character including numerals, punctuation, letters, and spaces.

The end of line is marked with a CR/LF pair as is typical of text files on a Windows system.

As an example, the following lines in the input file:

```
4Z<eol>
2 1Z<eol>
1 1Z<eol>
4Z<eol>
```

Would produce:

```
ZZZZ
  Z
 Z
ZZZZ
```

Note the space between the 2 and the 1 on the second line of the example input file. Space is a character that is repeated twice.

Write an RLE ASCII art decompressor. Your program should prompt the user for a filename of the compressed artwork, open the file (located in the current/same directory as program), and decompress the artwork to the console. Sample artwork files are provided on the competition resource page.

Sample session (bold is program-generated output):

```
In [7]: run prob5.py

Enter artwork filename: art.txt
ZZZZ
  Z
 Z
ZZZZ
```

Problem 6: RLE ASCII Art Compressor – 20 Points

Write an RLE ASCII Art Compressor that will generate the RLE-compressed version given "raw" ascii art per the compression scheme outlined in problem 5. Your program should prompt the user for a file (located in the current/same directory as program), compress the artwork located in the file, and display the compressed version to the console.

Ideally, your solution to problem 5 can take the output of this program as input and reproduce the original artwork. Sample artwork files are provided on the competition resource page.

Sample session (bold is program-generated output):

```
In [12]: run prob6.py

Enter artwork filename: z.txt
4Z
2 1Z
1 1Z
4Z
```

Problem 7: UNIX Time to Time and Date – 40 Points

Classically, UNIX-based systems keep track of time as the number of seconds elapsed since January 1, 1970 as a signed 32-bit integer. Of course, every day has exactly 86,400 seconds.

Write a program that will accept a numeric input of a positive UNIX time and produce a time and date string with hours, minutes, AM/PM, month, day, and year (e.g. HH:MM [AM|PM] MM/DD/YYYY). Time can be to the truncated minute (ignore seconds).

- Do not forget to account for leap years. Accounting for leap years is somewhat simplified by the fact that positive UNIX time spans only from 1970 to 2038, and there was no exception in year 2000.
- You should not account for time zones, thus the date and time you provide will be in UTC/GMT.
- The program should work for any valid positive UNIX time which is a signed 32-bit integer.
- Although technically valid times, your program does not need to work for negative time values.
- You may not use any library routines for manipulation of time or dates, such as converting UNIX time to a date string or for formatting date strings in general. All calculations for determining year, month, day, and time must appear in your source. **Source will be reviewed and rejected if deemed in violation.**
- There are numerous resources on the web for converting UNIX time. Use one for testing along with the examples shown below.

Sample session (bold is program-generated output):

```
In [13]: run prob7.py

Enter UNIX time: 0
12:00 AM   01/01/1970

In [14]: run prob7.py

Enter UNIX time: 1234567890
11:31 PM   02/13/2009

In [15]: run prob7.py

Enter UNIX time: 1111111111
01:58 AM   03/18/2005
```

Problem 8: Check My Digits – 40 Points

A check digit is present in many everyday numbers to provide an instant check for common typing errors such as single mis-keyed numbers or transposed digits. Consider the following check digit algorithm for 15-digit numbers (digit positions are 1s-based counting left to right):

- Digits 2,5,8,11 and 14 are doubled
- Digits 3,6,9,12 and 15 are tripled
- Any digits that have exceeded 9 are reformed into single digits by summing those digits
- All digits are added together
- The remainder of this sum is divided by 11 (sum modulo 11). If the remainder is 0, then the check digit is 0. If the remainder is non-zero, the value of the check digit is 11-remainder.
- If the check digit results in a value of 10, the single digit X is used to signify a value of 10

The final number used will be the initial 15 digit number plus the check digit. When verifying the composite number, the same procedure is used, but now the check digit is included with the sum (substituting a value of 10 for X, of course). Now when the sum is divided by 11, the remainder should be 0 if the number was keyed in correctly.

Digit # -->	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16			
Number -->	8	6	7	5	3	0	9	4	1	4	2	6	2	1	1				
Operation -->	↓	x 2	x 3	↓	x 2	x 3	↓	x 2	x 3	↓	x 2	x 3	↓	x 2	x 3				
Result -->	8	12	21	5	6	0	9	8	3	4	4	18	2	2	3		Sum ↓↓	Remainder ↓↓	Check Digit ↓↓
Single Digit -->	8	3	3	5	6	0	9	8	3	4	4	9	2	2	3		69	3	8

Figure 1. Summary of operations for check digit calculation.

Write a program that will prompt the user to enter a 15 or 16 digit number as a single string – not a series of separate entries. If the user enters a 15-digit number, the program should calculate and display the check digit. If the user enters a 16-digit number, the program should verify the check digit and display whether the number passes the check or not. You can assume that the user will enter a fifteen or sixteen digit number with proper digits 0-9 plus possibly X (or x) only as the 16th digit.

Sample session (bold is program-generated output):

```
In [2]: run prob8.py

Enter number (15 or 16 digits): 867530941426211
Check digit is:  8

In [3]: run prob8.py

Enter number (15 or 16 digits): 8675309414262118
Number passes check

In [4]: run prob8.py

Enter number (15 or 16 digits): 6875309414262118
Number fails check
```

Digits transposed

Problem 9: Simple Cipher – 40 Points

Many cryptographic ciphers rely on the bitwise XOR function at some point because it is a reversable function. That is, if A XOR B = C, then C XOR B = A. In this example, within the context of a cipher, A would be the plaintext message, B is the key, and C is the ciphertext. Actual ciphers vary in how large the key is, how the plaintext message is organized prior to application of the key, how the key is managed and modified during encryption, etc. Since these details can be a little daunting, the following simplified algorithm is proposed as an encryption / decryption scheme. A property of this algorithm is that it is symmetrical, that is, the algorithm applied to plain text will produce cipher text and vice versa. Another property of this algorithm is that the ciphertext is human readable, so we do not need to handle awkward binary or hexadecimal numbers; although as described, punctuation will appear in the cipher text. A third property is that it is not all that secure…

As a first step, the text input is effectively limited to upper case letters, spaces, most punctuation, and numerals. More specifically, ASCII values 32 (space) to 95 (underscore). To enhance robustness, convert any lowercase characters to upper case prior to encryption. The size of the input must be a multiple of five, therefore may need to be padded with spaces.

Each letter, is then reduced to numeric form by subtracting 32 from its ASCII value, thus, valid characters will now range from 0 to 63. This number can now be represented with just six bits. Starting at the beginning of the message, a group of five characters is then formed into a single 32-bit number, occupying the lower 30-bits. The most significant two bits are never used.

The key is then bitwise XORed with this grouping of characters. The result is the cipher text for this first group of five characters. The cipher text can then be output as human readable text by reversing the transformation described above (that is, separating the 30-bit result into five 6-bit numbers, adding 32 to each to arrive at a printable ASCII code).

For the next grouping of characters, we will modify the key by doing a rotating-left-shift over the 30-bits that are used to XOR the character data.

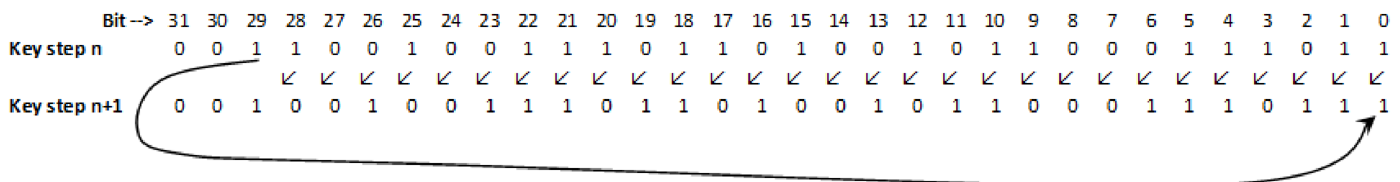

Figure 2. Key update algorithm. Key rotates left one bit, with 30th bit (bit 29) wrapping around to bit 0.

Implement this cipher. Create a program that will accept either plain text or cipher text along with a number to be used as the initial key. Ideally this number would be large (up to about 30 bits). The program shall then apply the algorithm stated above and display the resulting cipher or plaintext to the console. As stated above, your program should be able to accept lower case letters, but then convert them to uppercase to work with the algorithm. See sample session for example.

Plain Text -->	W	E	L	C	O	M	E	space	T	O
ASCII Value -->	87	69	76	67	79	77	69	32	84	79
Reduced -->	55	37	44	35	47	45	37	0	52	47
Binary -->	110111	100101	101100	100011	101111	101101	100101	000000	110100	101111
Combined -->	110111100101101100100011101111					101101100101000000110100101111				
Key -->	000000100001000101111111101101					000001000010001011111111011010				
XORed	110111000100101001011100000010					101100100111001011001011110101				
Split -->	110111	000100	101001	011100	000010	101100	100111	001011	001011	110101
Reduced -->	55	4	41	28	2	44	39	11	11	53
ASCII -->	87	36	73	60	34	76	71	43	43	85
Cipher Text -->	**W**	**$**	**I**	**<**	**"**	**L**	**G**	**+**	**+**	**U**

Figure 3. Encrypting "WELCOME TO." Initial key is decimal 8675309. Note key is shifted one bit left for second group of characters.

Sample session (bold is program-generated output):

```
In [39]: run prob9.py

Enter plain or cipher text: Welcome to MSOE

Enter key: 8675309
W$I<"LG++U"ID11

In [40]: run prob9.py

Enter plain or cipher text: W$I<"LG++U"ID11

Enter key: 8675309
WELCOME TO MSOE
```

All caps since input was converted to all caps prior to encryption.