ESC190H1S Winter 2022: Assignment 2

Due April 14, 2022 at 23:59 on Gradescope; up to 48 hour grace period where no late penalty will be incurred. No late submissions accepted after April 16, 2022 at 23:59.

Overview

Sandy and Cima grow paranoid that someone is trying to uncover the secret formula for their restaurant's signature dish, the Tasty Patty. They decide to encrypt their communications in order to evade their phantom spy. In the process, they will have to figure out how to perform simple data compression and identify evidence of data corruption. Anticipating a ton of memory mismanagement ahead, they turn to you for help coding some functions to put everything together - Good luck!

Code Submission

Carefully review the provided starter code. You will only be submitting `a2.c` for grading. Complete the required functions specified in `a2.c`, adding any additional helper functions and their prototypes to this file. You are permitted, but not required, to work with a partner on this assignment. Declare your partner on Gradescope - you do not need to work with your lab partner for this assignment.

Violation of any of these constraints may result in a grade of zero from the autograder.

- You **MUST NOT** modify `a2.h`.
- Your code **MUST** work when the defined `KEY` value in `a2.h` is changed.
- Your code **MUST** compile on the autograder to receive any credit.
- You **MUST NOT** modify the function signatures.

Additional Remarks

It is highly recommended that you read through the entire document and starter code before starting this assignment. Think carefully about where to allocate and free memory so that the code runs Valgrind clean. Test your code frequently (with Valgrind where appropriate) and ensure there are no issues before you write more code. Feel free to code the functions out of order.

Sliding into the QR codes

To disguise their communications, Sandy and Cima decide to encrypt their messages and send them as *SC codes*, a slight departure from the pre-existing QR code system. Message strings will be converted to encrypted SC codes by (1) performing bit-wise XOR encryption on the ASCII value of each character in the message, (2) converting the encrypted value to a 7-bit binary string of `'1'`s and `'0'`s, and (3) copying the encrypted strings generated for each character into a character array, following a specified SC code structure. As the SC codes pile up, Sandy and Cima realize they need a means of archiving and verifying the authenticity of their conversations... but first we must encrypt (and decrypt)!

Task 1: Character to binary, XOR, and back again

We will use **XOR encryption** to encrypt and decrypt characters. To encrypt a character, we perform a bit-wise XOR operation on the binary representation of the character's ASCII value with a key value. The key value will be the binary representation of the ASCII value of `KEY`, a character defined in `a2.h`. The result of the encryption is a new binary number. To recover the original character, we simply perform an XOR operation on the encrypted binary number using the same key value used for encrypting.

Recall binary representation of numbers from Lab 0. To encrypt the character `'C'` (ASCII value 67) using a key of `'S'` (ASCII value 83), we XOR the binary bits of 67 (1000011) and 83 (1010011) and get 0010000. To decrypt 0010000 and recover the character `'C'`, we XOR the bits of 0010000 with our key (character `'S'`, ASCII 83, binary 1010011).

```
        'C' (ASCII 67)          1000011
XOR     'S' (ASCII 83, key)     1010011
Encrypted:      (16)            0010000

        ?   (16, encrypted)     0010000
XOR     'S' (ASCII 83, key)     1010011
Decrypted:      (67)            1000011 → 'C'
```

First Bit	Second Bit	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Complete the following functions in `a2.c` to encrypt and decrypt the characters of a message:

```
int bitwise_xor(int value);
```

This function returns the result of performing a bit-wise XOR operation on `<value>` with the ASCII value of the `KEY` character defined in `a2.h`.

For example, `bitwise_xor(67)` should return the integer `16` if `KEY` is `'S'` (ASCII value 83) and `bitwise_xor(83)` should return the integer `83` if `KEY` is `'\0'` (ASCII value 0).

Input: `<value>` is an integer. You may assume `<value>` is a positive ASCII value.
Output: Return the integer result of performing a bit-wise XOR operation on `<value>` with the ASCII value of `KEY`, defined in `a2.h`.

```
char *xor_encrypt(char c);
```

This function generates a 7-digit string of `'1'`s and `'0'`s, representing the binary result of performing a bit-wise XOR operation on the ASCII value of `<c>` with the ASCII value of `KEY`, defined in `a2.h`.

For example, `xor_encrypt('C')` should return a pointer to `"0010000"` if `KEY` is `'S'` and `xor_encrypt('S')` should return a pointer to `"1010011"` if `KEY` is `'\0'`.

Input: `<c>` is a character.
Output: Return a pointer to a 7-digit string of `'1'`s and `'0'`s, representing the binary result of performing a bit-wise XOR operation on the ASCII value of `<c>` with the ASCII value of `KEY`, defined in `a2.h`.

```
char xor_decrypt(char *s);
```

This function deciphers which character is represented by `<s>`. Assume `<s>` represents an XOR encrypted character that must be decrypted with an XOR operation using `KEY`, defined in `a2.h`, as the key value.

For example, `xor_decrypt("0010000")` should return the character `'C'` if `KEY` is `'S'` and `xor_decrypt("1010011")` should return the character `'S'` if `KEY` is `'\0'`

Input: `<s>` is a pointer to a 7-digit string of `'1'`s and `'0'`s.
Output: Return the character obtained after evaluating the ASCII value resulting from performing a bit-wise XOR operation on the value represented by `<s>` with the ASCII value of `KEY`, defined in `a2.h`.

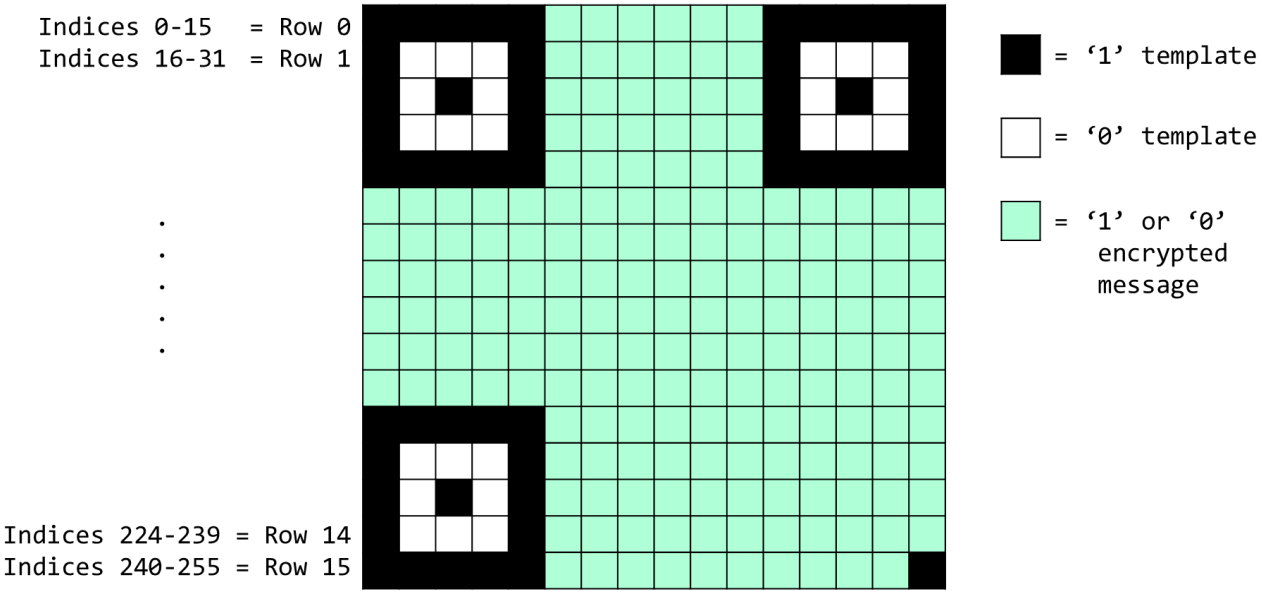
Task 2: Send SC codes

Now that we have functions to encrypt the individual characters of our message, we need to store the encrypted characters in an SC code template. The SC code template is as follows:

- The code is a single string of `'1'`s and `'0'`s, representing a 16x16 grid with one row stored after the other, sequentially. There should be 256 digits in the string for each SC code.

- The top-left, top-right, and bottom-left corners contain a 5x5 square of `'1'`s, an inner 3x3 square of `'0'`s, and a central `'1'`. The bottom-right corner should contain a `'1'`.
- The rest of the grid is dedicated to storing the encrypted characters of the message, starting from left to right, top to bottom. The first character of the message will be encrypted to a 7-digit string of `'1'`s and `'0'`s and stored at indicies 5-10 and 21 of the return string. The second character, after encryption, should then be stored at indices 22-26 and 37-38 of the return string. An encrypted null character indicates the end of the message and the remaining message region should be filled with `'0'`s. You may assume the message to be encrypted will fit in the SC code template.

The general layout of an SC code is visualized below.
An example of a filled-in SC code is included in Appendix A.



Complete the following functions in `a2.c` to generate and read SC codes:

```
char *gen_code(char *msg);
```

This function encrypts the characters of `<msg>` and stores the results in an SC code string.

Appendix A contains the output of `gen_code("Program in C!")` when the 256 digits of the return string are re-arranged into a 16x16 grid.

Input: `<msg>` is a pointer to a string representing the message to be encrypted.
Output: Return a pointer to a 256-digit string of `'1'`s and `'0'`s, representing an SC code containing the encrypted message.

```
char *read_code(char *code);
```

This function deciphers the message encoded in an SC code and generates a string containing the deciphered message.

When the SC code in Appendix A is passed to `scan_code()` as a single string input, a pointer to the string `"Program in C!"` should be returned.

Input: `<code>` is a pointer to a 256-digit string of `'1'`s and `'0'`s, representing an SC code containing an encrypted message.
Output: Return a pointer to a string containing the message obtained by deciphering the input SC code.

Task 3: Left on hex(adecimal)

After sending a flurry of SC codes back and forth, Sandy and Cima realize they need a more memory-efficient way to store their message history. They realize that they can compress their old SC codes by converting them from a 256-bit binary representation (base-2) to a 64-bit **hexadecimal** representation (base-16). In hexadecimal, each hex digit, or hex bit, is a coefficient of descending powers of 16. A hex bit could contain a value of 0-9 to represent a coefficient of 0-9 or a value of A-F to represent a coefficent of 10-15. To convert from binary to hexadecimal, we simply translate every four binary bits to a hex bit. To convert back to binary from hexadecimal, each hex bit is translated to four binary bits.

For example, the binary number `1111100000111111` would be represented in hexadecimal as `F84F`. Working with four binary bits at a time, binary `1111` corresponds to hexadecimal `F`, `1000` corresponds to `8`, and `0011` corresponds to `3`.

```
Binary        (Base-2)      1111 1000 0011 1111
Hexadecimal   (Base-16)       F    8    3    F
```

Complete the following functions in `a2.c` to compress and decompress an SC code:

```
char *compress(char *code);
```

This function generates a string representing the results of converting the binary digits of an SC code (`'1'`s and `'0'`s) to hexadecimal digits (`'0'`-`'9'`, `'A'`-`'F'`).

When the SC code in Appendix A is passed to `compress()` as a single string input,

the string `"F83F8D11AAF58C71FA3F0B27DCDD3DE6439530000000F8008800A8008800F801"` should be returned.

Input: `<code>` is a pointer to a 256-digit string of `'1'`s and `'0'`s, representing an SC code.
Output: Return a pointer to a 64-digit string containing the hexadecimal representation (`'0'`-`'9'`, `'A'`-`'F'`) of the binary digits in `<code>`.

```
char *decompress(char *code);
```

This function generates a string representing the results of converting the hexadecimal digits (`'0'`-`'9'`, `'A'`-`'F'`) of a compressed SC code to binary digits (`'1'`s and `'0'`s).

`decompress("F83F8D11AAF58C71FA3F0B27DCDD3DE6439530000000F8008800A8008800F801")` should return the SC code in Appendix A as a single 256-digit string of `'1'`s and `'0'`s.

Input: `<code>` is a pointer to a 64-digit string containing the hexadecimal representation (`'0'`-`'9'`, `'A'`-`'F'`) of a (compressed) SC code.
Output: Return a pointer to a 256-digit string of `'1'`s and `'0'`s, representing an SC code.

Task 4: Counting Corruption

Undeterred by their encrytion efforts, the phantom spy has hacked their way into Sandy's mobile device and has started overwriting some of Sandy's stored SC codes. To assess the extent of the data corruption, Cima suggests writing some code to count the number of changes made to Sandy's corrupted SC codes by measuring their **Levenshtein distance (LD)** from the original copy of SC codes stored in Cima's phone.

LD is a measure of similarity between two strings - the smaller the LD, the greater the similarity. To calculate the LD between two strings, we calculate the minimum number of character operations (insert, delete, or substitute) to transform one string into the other.

For two strings, C (length m) and S (length n), the LD between C and S is:

- m if S is an empty string, as m deletions are needed to transform C to S.
- n if C is an empty string, as n insertions are needed to transform C to S.
- equal to the LD between C[1:m] and S[1:n] if the first characters of each string, C[0] and S[0], are the same. If C[0] and S[0] are not the same, then the LD between C and S is:

$$1 + \text{the minimum of} \begin{cases} \text{the LD between C and S[1:n]} & \text{(Insertion of S[0] to C)} \\ \text{the LD between C[1:m] and S} & \text{(Deletion of C[0] from C)} \\ \text{the LD between C[1:m] and S[1:n]} & \text{(Substitution of C[0] with S[0])} \end{cases}$$

where we count 1 for the indicated operation and add it to the LD of the remaining substring(s). The LD is the *minimum* number of operations needed, so we add 1 to the minimum of the possible substring(s) to consider.

Two possible approaches to calculating LD are:

- The **recursive approach**, where substrings are passed to recursive calls. This approach is fairly straightforward, following the description above, but inefficient as the LD of the same substrings are calculated many times.
- The **dynamic programming** approach, where a temporary array is constructed with the LD of all substring combinations calculated once. In this approach, each element at index [row][column] of the array is the LD of the substrings, C[0:row] and S[0:column]. The top row is the number of insertions (0-n) starting from an empty string and the left-most column is the number of deletions (0:m) to generate an empty string.
  At each index [row][column] of the array, if the corresponding characters of the two substrings are equal, we assign the value from index [row-1][column-1] (no operation needed). Otherwise, we assign the value:

$$1 + \text{the minimum of} \begin{cases} \text{index[row-1][column-1]} & \text{(Substitution)} \\ \text{index[row-1][column]} & \text{(Deletion)} \\ \text{index[row][column-1]} & \text{(Insertion)} \end{cases}$$

  See Appendix B for some examples of arrays generated when using this approach to calculate the LD between pairs of strings. Notice that the LD between the two strings being considered is found in the bottom right corner element.

Complete the following function in `a2.c` to calculate the LD between two strings:

```
int calc_ld(char *sandy, char *cima);
```

This function calculates the Levenshtein distance between two input strings.

For example, `calc_ld("COMMENCE", "CODING")` should return `5` and `calc_ld("COMMENCE", "PROCRASTINATING")` should return `13`.

Inputs: `<sandy>` and `<cima>` are pointers to a two strings with unknown similarity.
Output: Return the Levenshtein distance between the strings pointed to by `<sandy>` and `<cima>`, as an integer.

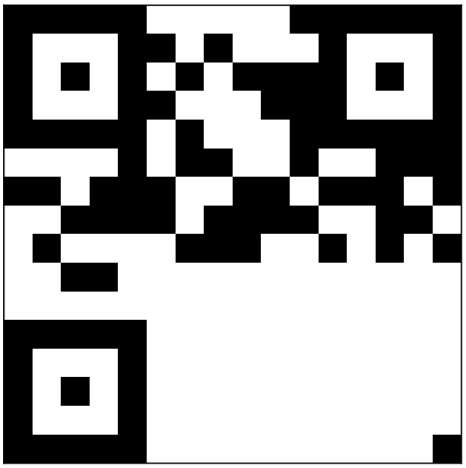
Appendix A

The SC code for the message `"Program in C!"`, encrypted with a `KEY` of `'S'` would look like this after re-arrangement into a 16x16 grid:

'1'	'1'	'1'	'1'	'1'	'0'	'0'	'0'	'0'	'0'	'1'	'1'	'1'	'1'	'1'	'1'
'1'	'0'	'0'	'0'	'1'	'1'	'0'	'1'	'0'	'0'	'0'	'1'	'0'	'0'	'0'	'1'
'1'	'0'	'1'	'0'	'1'	'0'	'1'	'0'	'1'	'1'	'1'	'1'	'0'	'1'	'0'	'1'
'1'	'0'	'0'	'0'	'1'	'1'	'0'	'0'	'0'	'1'	'1'	'1'	'0'	'0'	'0'	'1'
'1'	'1'	'1'	'1'	'1'	'0'	'1'	'0'	'0'	'0'	'1'	'1'	'1'	'1'	'1'	'1'
'0'	'0'	'0'	'0'	'1'	'0'	'1'	'1'	'0'	'0'	'1'	'0'	'0'	'1'	'1'	'1'
'1'	'1'	'0'	'1'	'1'	'1'	'0'	'0'	'1'	'1'	'0'	'1'	'1'	'1'	'0'	'1'
'0'	'0'	'1'	'1'	'1'	'1'	'0'	'1'	'1'	'1'	'1'	'0'	'0'	'1'	'1'	'0'
'0'	'1'	'0'	'0'	'0'	'0'	'1'	'1'	'1'	'0'	'0'	'1'	'0'	'1'	'0'	'1'
'0'	'0'	'1'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'1'	'1'	'1'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'0'	'0'	'0'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'0'	'1'	'0'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'0'	'0'	'0'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'1'	'1'	'1'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'1'

= template = message = leftover

On mobile device (just for fun, do not try to scan in real life):



After compression to a hexadecimal string:

`"F83F8D11AAF58C71FA3F0B27DCDD3DE6439530000000F8008800A8008800F801"`

Appendix B

`calc_ld("COMMENCE", "CODING")` should return 5.

	-	C	O	D	I	N	G
-	0	1	2	3	4	5	6
C	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
M	3	2	1	1	2	3	4
M	4	3	2	2	2	3	4
E	5	4	3	3	3	3	4
N	6	5	4	4	4	3	4
C	7	6	5	5	5	4	4
E	8	7	6	6	6	5	5

`calc_ld("COMMENCE", "PROCRASTINATING")` should return 13.

	-	P	R	O	C	R	A	S	T	I	N	A	T	I	N	G
-	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	1	1	2	3	3	4	5	6	7	8	9	10	11	12	13	14
O	2	2	2	2	3	4	5	6	7	8	9	10	11	12	13	14
M	3	3	3	3	3	4	5	6	7	8	9	10	11	12	13	14
M	4	4	4	4	4	4	5	6	7	8	9	10	11	12	13	14
E	5	5	5	5	5	5	5	6	7	8	9	10	11	12	13	14
N	6	6	6	6	6	6	6	6	7	8	8	9	10	11	12	13
C	7	7	7	7	6	7	7	7	7	8	9	9	10	11	12	13
E	8	8	8	8	7	7	8	8	8	8	9	10	10	11	12	13