Project 2: ASCII Cipher – Preston Hansen

I/O Demonstrations:

Note that all screenshots are of raw output from Mars pasted into a text document for easier readability. Also note that in all screenshots of part 2 or 3, the program (incorrectly) prompts for a string "to be encoded" in both phases. The functionality has not been modified in any way, but the prompt in the submitted code is slightly modified for ambiguity.

```
Enter a string to be encoded (max 20 characters): aaaabcddeeeeef
Enter a key (int): -28
New String: yzabdfhiklmnoq


Range of characters for decoded string:
 [a,f]
The most common character for the decoded string: e
Its frequency: 5
Range of characters for encoded string:
 [a,z]
The most common character for the encoded string: a
Its frequency: 1
Enter a string to be encoded (max 20 characters): yzabdfhiklmnoq
Enter a key (int): -28
New String: aaaabcddeeeeef


Range of characters for decoded string:
 [a,f]
The most common character for the decoded string: e
Its frequency: 5
Range of characters for encoded string:
 [a,z]
The most common character for the encoded string: a
Its frequency: 1
```

The first demonstration shows the output of part 2, which exhibits proper behavior for wrap-back (given a negative key), handling a key with absolute value greater than 26, displaying range, displaying frequency, and properly decoding the encoded message back to its original form.

```
Enter a string to be encoded (max 20 characters): thissequenceistoolong
Enter a key (int): 0
New String: tikvwjwbmwmpufhdecgga

Range of characters for decoded string:
 [c,u]
The most common character for the decoded string: e
Its frequency: 3
Range of characters for encoded string:
 [a,w]
The most common character for the encoded string: w
Its frequency: 3
Enter a string to be encoded (max 20 characters): tikvwjwbmwmpufhdecgga
Enter a key (int): 0
New String: thissequenceistoolong

Range of characters for decoded string:
 [c,u]
The most common character for the decoded string: e
Its frequency: 3
Range of characters for encoded string:
 [a,w]
The most common character for the encoded string: w
Its frequency: 3
```

The second output, also from part 2, shows that the program cuts off user input properly at 20 characters, handling of a key 0, and simpler demonstration of the (k, k+1…k+n) encoding specification.

```
Enter a string to be encoded (max 20 characters):
Enter a key (int): 0

Received empty string. Exiting...

-- program is finished running --
```

The third output shows that in parts 1-3, empty strings are handled separately by simply displaying an error and exiting.

```
Enter a string to be encoded (max 20 characters): zzzzyxxaab
Enter a key (int): 30
New String: defggghlmo


Range of characters for decoded string:
 [a,z]
The most common character for the decoded string: z
Its frequency: 4
Range of characters for encoded string:
 [d,o]
The most common character for the encoded string: g
Its frequency: 3
Enter a string to be encoded (max 20 characters): defggghlmo
Enter a key (int): 30
New String: zzzzyxxaab


Range of characters for decoded string:
 [a,z]
The most common character for the decoded string: z
Its frequency: 4
Range of characters for encoded string:
 [d,o]
The most common character for the encoded string: g
Its frequency: 3
-- program is finished running --
```

This output shows the previously described features, but also that large, positive keys, and wrapping “forward” from z to a also works properly.

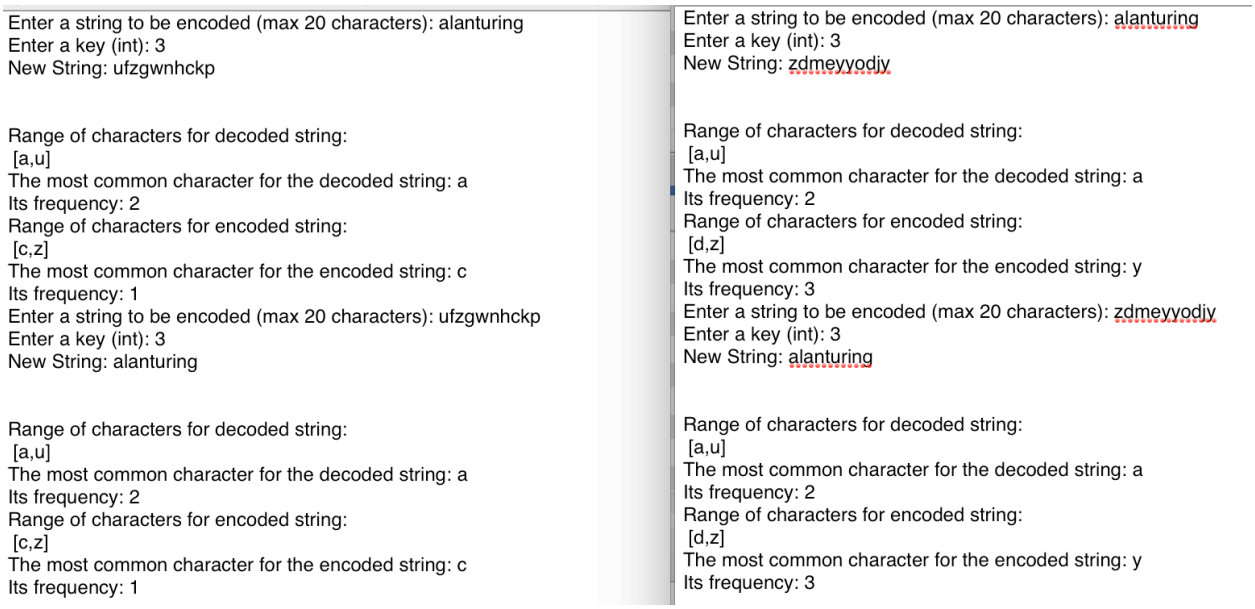
```
Enter a string to be encoded (max 20 characters): abcdefzzz
Enter a key (int): -3
New String: xyzabcwww

Range of characters for decoded string:
 [a,z]
Range of characters for encoded string:
 [a,z]
Enter an encoded string to be decoded: xyzabcwww
Enter a key (int): -3
New String: abcdefzzz

Range of characters for decoded string:
 [a,z]
Range of characters for encoded string:
 [a,z]

-- program is finished running --
```

This output is a simple demonstration of part 1, where the only significant difference is that frequencies are not shown, and the encoding is only shifted by k.



The final output demonstrates the functionality of part 3, where the encoding is based on a random sequence summed with k (by index). Within a run of the program, the sequence is preserved, so that an encoded string could be decoded back. Once the program is reloaded, the sequence is different, so the same string input will virtually never encode to the same output twice.

Program Features:

i. My program achieves levels 1 through 3. The specified functionality—i.e. encoding, decoding, character ranges, character frequencies, and wrap back/forward—all function properly assuming user input is within the problem spec. That is, for all lowercase inputs [a,z] and for 32 bit (signed) key inputs, the above features function properly (disregarding the prompt message which I previously noted).

ii. The most significant challenge I found in the project was debugging features which access memory. There was an error in my code when checking for the range of characters which would, only under certain circumstances, access adjacent memory values (and overwrite them), leading to strange and inconsistent behavior. Debugging this issue required careful use of breakpoints and isolating functions to test their behavior separately.

iii. N/A

Implementation Details:

i. To implement the encoding scheme, I accepted user input for the string and key, and then adjusted the key to a number within [-26, 26] by subtracting or

adding 26 appropriately, such that “shifting” behavior would remain unchanged. To encode, I iterated over the user input, and, assuming the current character was not a newline or null terminator, performed the encoding operation. For level 1, I simply added k to the character. For level 2, I added k plus the index of the character to the character (k, k+1, ..., k+n). For level 3, I generated a sequence of 20 pseudo-random 1-byte integers range [0, 255] such that in signed representation, approximately half would be positive, and half would be negative. I then added k and the integer with the corresponding index to the character. All of the operations for the decoding phase were equivalent, only using subtraction instead of addition.

In all of these processes, I checked after addition/subtraction of the characters if the value fell within the range of [a,z]. If not, I wrapped “back” or “forward” depending on the value. To do this, I took the difference between the value and ‘a’ or ‘z’. I then added this value to ‘a’-1 or ‘z’+1, and stored it as the shifted value if it fell within [a,z], otherwise, I repeated the process. For example, the character ‘a’ with a key of -1 in the encoding phase would map to ‘z’. Likewise, ‘z’ with a key of -1 would map to ‘a’ in the decoding phase, and so on.

To calculate the range of characters, I simply allocated 4 bytes to store one ASCII character for the biggest and smallest of the encoded and decoded strings. I then simply iterated over the string, and compared each character to the current min and max (initialized to the first character of the string).

To determine the frequencies of characters, I allocated four 26-byte sequences in memory, where each index corresponded to a letter of the alphabet, initialized to 0. For each character I encountered in the string, I subtracted ‘a’ from its value, and then incremented the value of its corresponding index. For example, the string “aaabf” would subtract ‘a’ from ‘a’, which would result in the 0^{th} index of the array being incremented by one (three times). Thus, the final array in memory would look like [3, 1, 0, 0, 0, 1, 0, 0, …]. I then iterated over the array and determined which index held the largest value. Once I had iterated through the whole array, I simply added the index value to ‘a’ and stored the resulting character in a separate byte, and stored its frequency in another byte. This process repeated four times (one for each encoded/decoded string).

As an optimization, I performed the range and frequency checks in one pass, meaning the operations to find range, frequency, and most common character were all linear in time complexity, and constant in space complexity.

ii. For levels 1 and 2, the encryption method is fairly weak. Any input which contains repeated characters would reveal the pattern of encoding,

especially if an onlooker was allowed to use the system itself, instead of just reading messages. For level 3, the encryption is more robust. The main weakness of my method is that given the range of [a,z], any multiple of 26 plus a given integer will map a character to the same value. That is, 'a' shifted by 26 or 52 would map to the same value. If two of the sequence values happened to be the same (or offset by a multiple of 26), and the characters at those indices were the same, then those characters would map to the same value. This is not necessarily a serious concern, however, since the values are randomly generated for each run of the program. Thus, there are 26^L ways that a single string length L could be mapped to an encoding. A brute force method of checking all possible permutations of 20-character strings against an encoded value would need to check $1.9928 * 10^{28}$ different strings. Checking one million strings per second with this method would take roughly $6.3 * 10^{14}$ years. If a third party was only looking at the encoded strings, there would be no feasible way to decrypt a given message without knowing the encoding sequence.

iii. Given another month, I would improve the encryption scheme by requiring the user to enter a sequence as their key, not just an integer. If this sequence required at least one ASCII symbol, number, and upper/lower case character, I would then develop an algorithm to generate a key at runtime based on the user input. This could be used to uniquely encrypt sequences for a given user, while preserving the ability to encode/decode across many sessions. Alternatively, I could purchase some books about cryptography or discrete math and implement a simpler version of RSA.

If I was given 3 weeks to redo this project, I would most likely reuse the same functionalities, but trade space/time complexity for readability. For instance, I would separately calculate character ranges, frequencies, etc. instead of calculating them in one pass to make the program easier to modify and debug.