CSE 232

Programming Project 04

Assignment Overview
This assignment will give you more with functions and introduce the use of the string data type.

Background
You probably know about programs like zip. The basic idea of zip is to take a file and compress it, that is change the representation of the file such that it occupies less space without losing any information in the process. zip and its relatives are known as a kind of lossless compression. There are other kinds of compression, called lossy compression, that will sacrifice full accuracy for better compression of the file. We want to look at a simple compression algorithm called run-length encoding.

Run-length encoding
Run-length encoding (see https://en.wikipedia.org/wiki/Run-length_encoding) is a fairly simple kind of compression. In its simplest form, which we are looking at here, you take a sequence of repeated, single characters and turn those characters into a representation that takes less space. Taking less space, using fewer characters than the original, is the goal here so it is important to keep that in mind.

Here's an example of how we would like to run-length encode a string:

`aaaaaabbbbbccccdddeef` → `:ca:bb:acdddeef`

The ratio of compression is 21 to 15 or a reduction to15/21 (71.4%) of its original size.

How do we interpret the above encoding? Let's discuss.

Our encoding
Let's look at one piece of our encoding:

`aaaaaa` → `:ca`

Every encoding uses three characters. The first is a marker character to indicate that what follows was encoded. The second is a count of how many characters there are (more on that in a moment), and the third is the character that is repeated.

So how does the character 'c' represent 6 (the number of a's being encoded). Well, we could use decimal integers (something like :6a) but as soon as there are more than 9 characters in a sequence the encoding requires an extra character (something like :22a for a sequence of 22 a's). We are trying to be as conservative as possible to get the best compression, so we develop a scheme to represent decimal values as individual characters. Here is what we will use: character above, integer value below:

```
a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
```

First, why start at 4? Since an encoding uses three characters, there is no point in encoding any sequence that is 3 characters or less. Thus we pass those characters through without encoding. Since the first count we would encode is 4 characters, we use 'a' to represent a count of 4. Second, why stop at 55/capital Z. We could indeed go further and use all kinds of weird characters (like ~, $, *, etc.) to represent a count which would get us into the 100's, but we thought 4-55 is a good enough range for this project.

Furthermore, since we will not encode a sequence of 3 or less, we just pass them through unencoded. For sequences of 2 or less it is actually more efficient since 3 characters are required for an encoding. Going back then:

`aaaaaabbbbbccccdddeef` → `:ca:bb:acdddeef`

Means 6 a's, followed by 5 b's, followed by 4 c's, followed by 3 d's (no encoding), followed by 2 e's (no encoding) followed by 1 f (no encoding)

Project Description / Specification

Warning

Again, in this project as in the last we provide *exactly* our function specifications: the function name, its return type, its arguments and each argument's type. We will often provide you with a `main` program to include in your file (or a separate main program file when we start to develop mult-file programs). Do not change the function declarations!

function: `main`**:**

We provide in the Mimir project a starter file proj04.cpp without the functions. We provide this as a test rig, the kind you should be writing eventually for yourself. When run the main test each function individually based on input. Place your function code in the file marked by a comment. Feel free to experiment and test as you like, but remember to put the main code back to its original setup when you turn the code in. The TAs will be looking to see that the main you turn in is unchanged.

function `encode_sequence`:

- two arguments:
  - string consisting of all the same chars (no error checking)
  - char which is used to indicate an encoding
- return is a string

Encodes the provided string into a run-length encoding for the string of the same characters.

`encode_sequence("aaaaaaaaaa", '+')` → `"+ga"`

function: `encode`**:**

- two arguments:
  - string to encode
  - char which is used to indicate an encoding
- return is a string

Does the run-length encoding for the entire string. Should use `encode_sequence`.

`encode("aaaaaaaaaattttttddeef", ':')` → `:ga:ctddeef`

function: `decode_sequence`:

- two arguments
  - string consisting of a three character encode sequence(no error checking)
  - char which is used to indicate an encoding

- return is string

Converts the encode sequence into its original form.
`decode_sequence("+ga", '+')` → `"aaaaaaaaaa"`

function: `decode`:

- two arguments:
  - string to decode
  - char which is used to indicate an encoding
- return is string

Does the run-length decoding for the entire string. Should use `decode_sequence`.

function: `reduction`:

- two arguments (in this order):
  - original string
  - the run-length encoded string
- returns double

Returns the amount of reduction (the ratio of sizes, encoded to original).
`reduction("aaaaaaaaaa", "+ga")` → `0.30`

Deliverables

`proj04/proj04.cpp` -- your source code solution (*remember to include your section, the date, project number and comments in this file*).

1) Be sure to use "proj04.cpp" for the file name and to have it in a directory called proj04
2) Electronically submit a copy of the file to Mimir for testing

Discussion/Hints