UNIVERSITY of FLORIDA
Herbert Wertheim
College of Engineering

COP3504C

P1: RLE with Images

Overview

In this project students will develop routines to encode and decode data for images using run-length encoding (RLE). Students will implement encoding and decoding of raw data, conversion between data and strings, and display of information by creating procedures that can be called from within their programs and externally. This project will give students practice with loops, strings, arrays, methods, and type-casting.

Run-Length Encoding

RLE is a form of lossless compression used in many industry applications, including imaging. It is intended to take advantage of datasets where elements (such as bytes or characters) are repeated several times in a row in certain types of data (such as pixel art in games). Black pixels often appear in long "runs" in some animation frames; instead of representing each black pixel individually, the color is recorded once, following by the number of instances.

For example, consider the first row of pixels from the pixel image of a gator (shown in **Figure 1**). The color black is "0", and green is "2":

Flat (unencoded) data: 0 0 2 2 2 0 0 0 0 0 0 2 2 0

Run-length encoded data: 2 0 3 2 6 0 2 2 1 0.

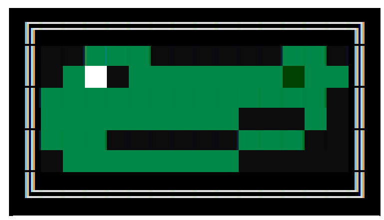


Figure 1 – Gator Pixel Image

The encoding for the entire image in RLE (in hexadecimal) – width, height, and pixels - is:

1 E |1 6 2 0 3 2 6 0 2 2 2 0 1 2 1 F 1 0 7 2 1 A F 2 1 0 9 2 3 0 1 2 1 0 3 2 6 0 3 2 3 0 8 2 5 0
\W/ \H/ \---PIXELS---/

Image Formatting

The images are stored in **uncompressed / unencoded** format natively. In addition, there are a few other rules to make the project more tractable:

1. Images are stored as an array of bytes, with the first two bytes holding image width and height.
2. Pixels will be represented by a number between 0 and 15 (representing 16 unique colors).
3. No run may be longer than 15 pixels; if any pixel runs longer, it should be broken into a new run.

For example, the chubby smiley image (**Figure 2**) would contain the data shown in **Figure 3**.

Figure 2

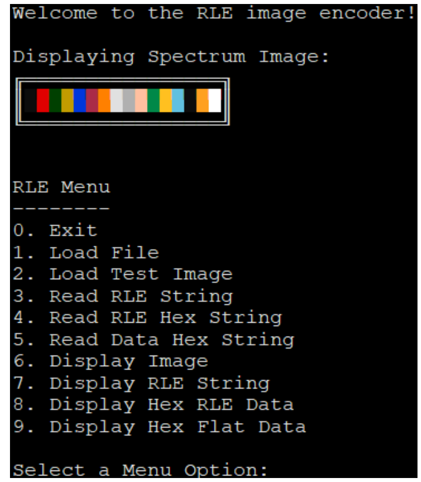
```
88 0B BB BB B0 BB BB BB
BB BB 0B B0 BB BB BB BB
BB BB 0B B0 BB BB B0 0B
BB BB BB BB BB 0B BB BB
B0
```

Figure 3 – Data for "Chubby Smiley"

NOTE: <u>Students do not need to work with the image file format itself</u> – they only need to work with byte sequences and encode or decode them. Information about image formatting is to provide context.

Requirements

Student programs must present a menu when run in standalone mode and must also implement several methods, defined below, during this assignment.



Standalone Mode (Menu)

When run as the program driver via the **`main()`** method, the program should:

1) Display welcome message
2) Display color test (**`console_gfx.TEST_RAINBOW`**)
3) Display the menu
4) Prompt for input

<u>Note: for colors to properly display, it is highly recommended that student install the "CS1" theme on the project page if they have not done so.</u>

There are five ways to load data into the program that should be provided and four ways the program must be able to display data to the user.

Loading a File

Accepts a filename from the user and invokes **`console_gfx.load_file`**`(`*`filename`*`: str)`:

```
Select a Menu Option: 1
Enter name of file to load: testfiles/uga.gfx
```

Loading the Test Image

Loads **`console_gfx.TEST_IMAGE`**:

```
Select a Menu Option: 2
Test image data loaded.
```

Reading RLE String

Reads RLE data from the user in decimal notation with delimiters (smiley example):

```
Select a Menu Option: 3
Enter an RLE string to be decoded: 28:10:6B:10:10B:10:2B:10:12B:10:2B:10:5B:20:11B:10:6B:10
```

Reading RLE Hex String

Reads RLE data from the user in hexadecimal notation **without** delimiters (smiley example):

```
Select a Menu Option: 4
Enter the hex string holding RLE data: 28106B10AB102B10CB102B105B20BB106B10
RLE decoded length: 66
```

Reading Flat Data Hex String

Reads raw (flat) data from the user in hexadecimal notation (smiley example):

```
Select a Menu Option: 5
Enter the hex string holding flat data: 880bbbbbb0bbbbbbbbbb0bb0bbbbbbbbbbbb0bb0bbbbb00bbbbbbbbbbb0bbbbbb0
Number of runs: 18
```

Displaying the Image

Displays the current image by invoking the **`console_gfx.display_image`**`(`*`imageData`*`: bytes)` method.

Displaying the RLE String

Converts the current data into a human-readable RLE representation (with delimiters):

```
Select a Menu Option: 7
RLE representation: 28:10:6b:10:10b:10:2b:10:12b:10:2b:10:5b:20:11b:10:6b:10
```

Note that each entry is 2-3 characters; the **length** is always in <u>decimal</u>, and the **value** in <u>hexadecimal</u>!

Displaying the RLE Hex Data

Converts the current data into RLE hexadecimal representation (**without** delimiters):

```
Select a Menu Option: 8
RLE hex values: 28106b10ab102b10cb102b105b20bb106b10
```

Displaying the Flat Hex Data

Displays the current raw (flat) data in hexadecimal representation (**without** delimiters):

```
Select a Menu Option: 9
Flat hex values: 880bbbbbb0bbbbbbbbbb0bb0bbbbbbbbbbbb0bb0bbbbb00bbbbbbbbbbb0bbbbbb0
```

Module Functions

Student modules are **required** to provide **all of the following functions** with the **defined behaviors**. We recommend completing them in the following order:

1. `count_runs(flatData: iterable) -> int`
   Returns number of runs of data in an image data set; double this result for length of encoded (RLE) byte array.

   Ex: `count_runs([15, 15, 15, 4, 4, 4, 4, 4, 4])` yields integer `2`.

2. `to_hex_string(data: iterable) -> str`
   Translates data (RLE or raw) a hexadecimal string (without delimiters). This method can also aid debugging.

   Ex: `to_hex_string([3, 15, 6, 4])` yields string `"3f64"`.

3. `encode_rle(flat_data: iterable) -> bytes`
   Returns encoding (in RLE) of the raw data passed in; used to generate RLE representation of a data.

   Ex: `encode_rle([15,15,15,4,4,4,4,4,4])` yields `b'\x03\x0f\x06\x04'` (i.e., `[3, 15, 6, 4]`).

4. `get_decoded_length(rle_data: iterable) -> int`
   Returns decompressed size RLE data; used to generate flat data from RLE encoding. (Counterpart to #2)

   Ex: `get_decoded_length([3, 15, 6, 4])` yields integer `9`.

5. `decode_rle(rle_data: iterable) -> bytes`
   Returns the decoded data set from RLE encoded data. This decompresses RLE data for use. (Inverse of #3)

   Ex: `decode_rle([3, 15, 6, 4])` yields `b'\x0f\x0f\x0f\x04\x04\x04\x04\x04\x04'`.

6. `string_to_data(data_string: str) -> bytes`
   Translates a string in hexadecimal format into byte data (can be raw or RLE). (Inverse of #1)

   Ex: `string_to_data("3f64")` yields `b'\x03\x0f\x06\x04'` (i.e., `[3, 15, 6, 4]`).

7. `to_rle_string(rleData: iterable) -> str`
   Translates RLE data into a human-readable representation. For each run, in order, it should display the run length in ***decimal*** (1-2 digits); the run value in ***hexadecimal*** (1 digit); and a delimiter, ':', between runs. (See examples in standalone section.)

   Ex: `to_rle_string([10, 15, 6, 4])` yields string `"10f:64"`.

8. `string_to_rle(rleString: str) -> bytes`
   Translates a string in human-readable RLE format (with delimiters) into RLE byte data. (Inverse of #7)

   Ex: `string_to_rle("10f:64")` yields `b'\x0a\x0f\x06\x04'` (i.e., `[10, 15, 6, 4]`).

Submissions

NOTE: Your output must match the example output *exactly*. If it does not, ***you will not receive full credit for your submission***!

File: rle_program.py
Method: Submit on ZyLabs

Do not submit any other files!