

ASSIGNMENT

Course Code	CSC212A
Course Name	Data Communication
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No	17ETCS002159
Semester/Year	04/2019
Course Leader/s	Dr. Rinki Sharma

Declaration Sheet			
Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	04/2019
Course Code	CSC212A		
Course Title	Data Communication		
Course Date		to	
Course Leader	Dr. Rinki Sharma		
<p>Declaration</p> <p>The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.</p>			
Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			
Signature of the Course Leader and date		Signature of the Reviewer and date	

Declaration Sheet	ii
Contents	iii
Question No. 1	4
A 1.1 Benefits and limitations of Source-Channel separation theorem:	4
A 1.2 Significance and applications of JSCC:.....	5
A 1.3 Conclusions:	5
Question No. 2	6
B 1.1 Introduction:.....	6
B 1.2 Algorithm/Flow Chart of RLE Computation:	6
B 1.3 Computation of compressed value of considered data using RLE compression:	7
B 1.4 Conclusion:	8
Question No. 3	9
B 2.1 Introduction:.....	9
B 2.2 Implementation of RLE:.....	9
B 2.3 Testing of RLE implementation and explanation of obtained output:	12
B 2.4 Conclusion:	13

Solution to Question No. 1 Part A:

A 1.1 Benefits and limitations of Source-Channel separation theorem:

Shannon in (1948) with his groundbreaking work "A Mathematical Theory of Communication". Here, he introduced the concept of channel capacity and proved that as long as the transmission rate is below the channel's capacity, reliable communication with a non-zero, but arbitrarily small error rate is possible. Furthermore, he presented a theorem stating that a source with entropy H can be reliably transmitted over a channel with capacity C as long as $H \leq C$.

In principle this can be achieved by first applying a source coder that reduces the rate of the source down to the minimum, called the entropy H , and subsequently applying a channel code. In the receiver, the channel decoder, which is unaware of the type of source, outputs the most probable codeword to the source decoder. Finally, the source coder reconstructs the source without any knowledge of the channel statistics. This independence between source and channel coder is the reason why this theorem is also known as the separation theorem; the simplifications it provides makes it prevalent in the communication community.

However, this theorem does not hold in every situation. In those cases where it breaks down, there are usually systems with combined or joint source-channel coding that perform better.

The fact that we can perform source coding independently of channel coding and vice versa simplifies the construction of the system since we can optimize the coders separately and still achieve optimality. Also, having either a different source or channel, we can change the affected coder while leaving the other unchanged. However, there are some drawbacks with this approach.

First of all, we have to allow infinite complexity and delay in the coders in order to reach optimality. Evidently this is problematic for real-time communication.

Secondly, the theorem is no longer valid for non-ergodic and multi-user channels, and in those cases, we no longer have an optimal system.

Thirdly, such systems tend to break down completely when the channel quality falls under a certain threshold, and the channel code is no longer capable of correcting the errors. This phenomenon is often referred to as the threshold effect.

For non stationary sources the source– channel partition coding theorem takes an other shape and we have to utilize ideas like "entirely commanding" and "control."

Thus, these systems are not robust with respect to changing channel qualities. Knowing that wireless channels have fluctuating channel qualities and high bit-error rates, trying to evade (or at least reduce) the threshold effect should be important when designing wireless communication systems.

A 1.2 Significance and applications of JSCC:

In a joint source-channel coding scheme, a single mapping is used to perform both the tasks of data compression and channel coding in a combined way, rather than performing them separately. Usually for simple iid sources and channels, separation of the tasks is information theoretically optimal.

Designing a communication system using joint source–channel coding in general makes it possible to achieve a better performance than when the source and channel codes are designed separately, especially under strict delay-constraints. The majority of work done in joint source-channel coding uses a discrete channel model, corresponding to an analog channel in conjunction with a hard decision scheme.

The performance of such a system can however be improved by using soft decoding at the cost of a higher decoding complexity. An alternative is to quantize the soft information and store the pre-calculated soft decision values in a lookup table.

Applications:

Current work in joint source and channel code design combines techniques for optimizing source codes to match channel noise characteristics with techniques for optimizing channel codes to match source code characteristics into a single joint design aimed at minimizing the end-to-end distortion experienced in the communication system as a whole.

The most important factor in that joint design seems to be the bit allocation between the source coding bits (used to describe the original data string) and the channel coding bits (used to protect the source coding bits from channel errors). The resulting codes may be employed on a wide range of channels (e.g., AWGN channels and Rayleigh fading channels with and without receiver side information) with significant performance benefits.

A 1.3 Conclusions:

- Source–channel separation does not hold in general for multiuser channels
- Joint source–channel coding schemes that utilize the correlation between the sources for cooperative transmission

When operating under a delay-constraint on a time-varying channel, it is generally no longer optimal to regard the two coders separately and we have to jointly optimise the source coder and the channel coder and the result is some sort of joint source-channel coding (JSCC). This is a rather loose label that encompasses all coding techniques where the source and channel coders are not entirely separated.

Solution to Question No. 1 Part B:

B 1.1 Introduction:

Data files frequently contain the same character repeated many times in a row. For example, text files use multiple spaces to separate sentences, indent paragraphs, format tables & charts, etc. Digitized signals can also have runs of the same value, indicating that the signal is not changing. For instance, an image of the nighttime sky would contain long runs of the character or characters representing the black background. Likewise, digitized music might have a long run of zeros between songs. Run-length encoding is a simple method of compressing these types of files.

RLE works by reducing the physical size of a repeating string of characters. This repeating string, called a run, is typically encoded into two bytes. The first byte represents the number of characters in the run and is called the run count. In practice, an encoded run may contain 1 to 128 or 256 characters; the run count usually contains as the number of characters minus one (a value in the range of 0 to 127 or 255). The second byte is the value of the character in the run, which is in the range of 0 to 255, and is called the run value.

B 1.2 Algorithm/Flow Chart of RLE Computation:

Algorithm for RLE Encoding

Arguments: str, length

Step 1: Start

Step 2: Declare encoded_output, prev_char = "\0", start, end, curr, curr_count = 1

Step 3: start = str

Step 4: end = str + length

Step 5: for curr = start; curr <= end ; curr++

Step 5.1: if curr != prev_char

Step 5.1.1: encoded_output = concatenate(encoded_output, prev_char)

Step 5.1.2: if curr_count > 1

Step 5.1.2.1 encoded_output = concatenate(encoded_output, curr_count)

Step 5.1.2: prev_char = curr

Step 5.1.3: current_count = 1

Step 5.2 else curr_count++

Step 6: return encoded_output

Step 7: Stop

B 1.3 Computation of compressed value of considered data using RLE compression:

Let's assume an input and compute the RLE for it,

INPUT : AABBBBBBCDEFFGGHIJKLMNOPPPPPPPPPPPPPPPPPPPPP

Using the Algorithm defined in B1.2

start = A (the memory location to the first character in INPUT)

end = P (actually the memory location to the last character in INPUT)

prev_char = ''

encoded_output = ''

current_count = 1

The loop runs from A to P, or from the start of the string to the end of the string

A != "", hence it is added to the encoded output,

encoded_output = "A"

Taking the next character from the input, A == 'A', current_count is incremented

current_count = 2

Taking the next character from the input A != B, since the current_count is > 1, it is added to the encoded output, and current_count is reset to 1

encoded_output = "A2"

Taking the next character from the input B == 'B', current_count is incremented

current_count = 2

This goes on for 6 times until we hit the next different character which is C, at this point the encoded output looks like,

encoded_output = "A2B6"

Taking the next character from input, C != B, and as the current_count not greater than 1, the count is not added to the encoded_output.

encoded_output = "A2B6C"

If we keep this algorithm running the encoded_output becomes,

OUTPUT: A2B6CDEF2G2HIJKLMNOP21

Let's try for another set of input, in this we do not have any repetition of character at all.

INPUT : SATYAJITGHANA

Since there is no repetition of characters continuously, the current count value stays at 1, and the count is not added to the output, hence in this case the output is same as that of the input.

OUTPUT : SATYAJITGHANA

But why aren't we adding the count to the output? this is because if the text to be encoded does not have any repetition, then the RLE would be double of the input, which is unnecessary, this can be avoided by not appending the count if its less than or equal to 1.

B 1.4 Conclusion:

In 'lossless compression', the codecs keep all of the information about a file. The compressed file, once decompressed, can be reconstructed so it is exactly like the file before it was compressed, with no loss of any information at all.

RLE works by looking through the data in a file and identifying repeating strings of characters, called a 'run'. The run is then encoded into a small number of bytes, usually two. The first byte, called the 'current_count', holds the number of characters in the run. The prev_character keeps track of the previous character in the run, every time this character is different the character is dumped to the output.

RLE algorithms are fast and simple. How well they compress data depends upon what is being encoded. The question here is about text, but RLE is mostly used in images, Let's suppose you are encoding a picture of a page in a book that you've just scanned. If the page is mostly white then RLE will compress the file very well because there will be lots of runs of the same ASCII code for white. If the page is mostly a busy colour photo, then there will be far less runs of the same ASCII colour code.

Solution to Question No. 2 Part B:

B 2.1 Introduction:

As we have already defined the RLE Algorithm in B1, the same has to be implemented in a program. The algorithm now is:

Step 1: Start

Step 2: Take input from the User from CLI or STDIN

Step 3: Compute RLE for the String using RLE_ENCODE Algorithm

Step 4: Display the encoded String

Step 5: Stop

That is just to make the program more interactive, STDIN is chosen as input since most LINUX/UNIX commands use STDIN and if large files have to be encoded they will definitely be using STDIN, makes the code more standardized.

B 2.2 Implementation of RLE:

data_compression.c

```
//  
// Created by shadowleaf on 09-Feb-19.  
//  
  
#include "data_compression.h"  
#include "vector.h"  
#include "debug_helper.h"  
#include <string.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
//#define DEBUG 0  
  
/* rle_encode : returns the Run-Length encoding of the string  
 * str : the string to be encoded  
 * length : length until which to be encoded */  
string rle_encode(string str, int length) {  
    /* create a empty placeholder for the encoded string */  
    char encoded_output[10000] = "\0";  
  
    /* to store the previous character  
     * assuming when there is empty input the prev_char is empty as well*/  
    char prev_char[2] = "\0";  
  
    /* pointers to string positions
```

```

    * encode the string from beginning to length of the string passed
    * as an argument */
char* start = str;
char* end = str + length;
char* curr;
/* to store the current count of the prev_char
   * initially empty hence empty count = 1*/
int curr_count = 1;

/* Iterative method to go through the entire string */
for (curr = start; curr <= end ; curr++) {
    if (*curr != prev_char[0]) {
        strcat(encoded_output, prev_char);
        /* store the repetitions only if the count is greater than 1
           * hence the length of a string with no repetition is not
increased */
        if (curr_count > 1) {
            char string_count[1000] = "\0";
            sprintf(string_count, "%d", curr_count);
            strcat(encoded_output, string_count);
        }
#ifdef DEBUG
        printf("%c : %d ", prev_char[0], curr_count);
#endif
        /* reset the prev_char and curr_count */
        prev_char[0] = *curr;
        curr_count = 1;
    } else {
        curr_count++;
    }
}

/* return the output with a new memory location */
return new_string(encoded_output);
}

```

data_compression.h

```

#ifndef RLE_DATA_COMPRESSION_H
#define RLE_DATA_COMPRESSION_H

#include "input_helper.h"

string rle_encode(string str, int length);

#endif //RLE_DATA_COMPRESSION_H

```

main.c

```

#include <stdio.h>
#include <string.h>

```

```

#include "data_compression.h"
#include "debug_helper.h"

void help();
string prog_name;
/* argc : argument count
 * argv : argument values*/
int main(int argc, char** argv) {
#ifdef DEBUG
    string str = "TEST STRING AAABBBCCCDEFFFF";
#endif
    prog_name = argv[0];
    if (argc == 3) {
        switch(*(argv[1]+1)) {
            case 's': {
                string encoded = rle_encode(argv[2], (int) strlen(argv[2]));
                printf("%s\n", encoded);
            } break;
            default: {
                help();
            }
        }
    } else if (argc == 1) {
        string str;
        while (get_line(&str) >= 0) {
            string encoded = rle_encode(str, (int) strlen(str));
            printf("%s", encoded);
        }
    } else {
        help();
    }
    return 0;
}

void help() {
    printf("Run-Length Encoder\n"
        "USAGE : %s -s <string>\n", prog_name);
}

```

Explanation:

The Program consists of taking the input from the user either from the command line, or from STDIN, the input is saved in a string of arbitrary length. The command line arguments uses the flag '-s' to specify the string to the user, and the STDIN is every line is taken as input and the corresponding compressed RLE is calculated and dumped to STDOUT.

The main function here manages the input and output, and the library function `rle_encode` encodes the string given to it.

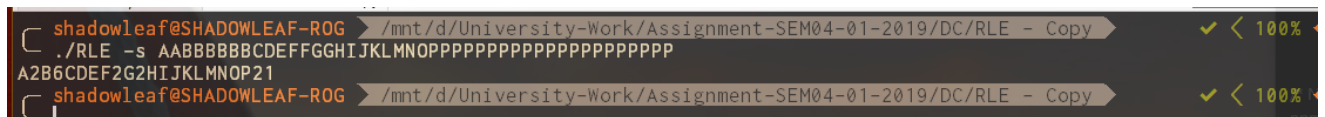
`rle_encode` takes in two arguments, the string and the length of the string upto which it is to be compressed. The logic of the program is that we keep track of the length of the current input symbol, and as soon as we hit a different symbol, we dump the count to the `encoded_output`, initially the `encoded_output` is an empty string.

The program starts by looking at the locations of the start and the end of the string, then it iterates over the string from beginning to end looking at every character.

As we see repeating characters the count is kept track of and is dumped as soon as the symbol changes, after this the count variable is reset to 1, and the process continues. We also use the fact that if the count is 1 then there is no use in adding it to the output since it will only increase the length of the `encoded_output`, which is undesirable, instead of compressing the string it will be performing the opposite.

After we've iterated over all the characters in the input, the `encoded_output` is returned from the function.

B 2.3 Testing of RLE implementation and explanation of obtained output:

A terminal window showing a command prompt. The user enters the command `./RLE -s AABBBBBBCDEFFGGHIJKLMNOPPPPPPPPPPPPPPPPPPPPP`. The output is `A2B6CDEF2G2HIJKLMNOP21`. The terminal background is dark with light-colored text. There are some status indicators on the right side of the terminal window, such as a green checkmark and a percentage.

```
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/Assignment-SEM04-01-2019/DC/RLE - Copy ✓ < 100%  
./RLE -s AABBBBBBCDEFFGGHIJKLMNOPPPPPPPPPPPPPPPPPPPPP  
A2B6CDEF2G2HIJKLMNOP21  
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/Assignment-SEM04-01-2019/DC/RLE - Copy ✓ < 100%
```

The explanation for this output is similar to that of the manual algorithm run for the input, We've taken the same input as of B1 so that it is comparable.

The input string here is `AABBBBBBCDEFFGGHIJKLMNOPPPPPPPPPPPPPPPPPPPPP`, which has many repetitions of characters.

The Program first takes the input from the Command Line Interface for simplicity, The string taken from input is then used as an argument for `rle_encode`, along with the length of the string.

`rle_encode` for the given input works like this:

Each character count is taken sequentially, and as soon as we hit a different character, the previous character is dumped to the output along with the character count, only if the count is greater than 1, this is to save the space when there are mostly unique characters in the input. This ensures that the algorithm does efficient compression.

The character count sequentially is as follow:

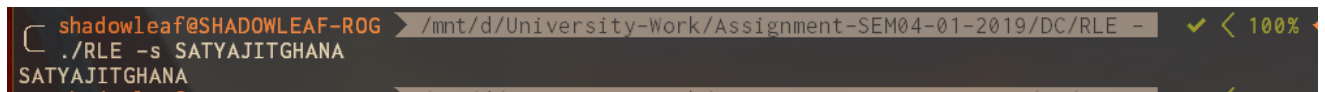
A: 2, B: 6, C: 1, D: 1, E: 1, F: 2, G: 2, H: 1, I: 1, J: 1, K: 1, L: 1, M: 1, N: 1, O: 1, P: 21

The characters which have count greater than 1 are dumped to the output along with their count, other characters are simply dumped without the count 1.

This gives the output as:

A2B6CDEF2G2HIJKLMNOP21

Which is same as that of the manual computation

A terminal window screenshot with a dark background. The prompt is 'shadowleaf@SHADOWLEAF-ROG'. The command entered is './RLE -s SATYAJITGHANA'. The output shown is 'SATYAJITGHANA'. The terminal title bar shows the path '/mnt/d/University-Work/Assignment-SEM04-01-2019/DC/RLE' and a progress indicator '100%'.

The character count sequentially for this is:

S: 1, A: 1, T: 1, Y: 1, A: 1, J: 1, I: 1, T: 1, G: 1, H: 1, A: 1, N: 1, A: 1

As we can see that all of them have a count of one, hence the output here is dumped same as that of the input.

SATYAJITGHANA

B 2.4 Conclusion:

Run-Length Encoding is an algorithm that exploits repetitive characters in sequence in a data by encoding it with a string that consists of the sum of the number of character looping that occurs, followed by a repeating character. Generally, optimal RLE algorithm is used on files that have characters that tend to be homogeneous. Therefore, if the algorithm is used universally then it is necessary to do the grouping or transformation of similar characters / symbols.

RLE doesn't work well with randomized data, its input is mapped directly to the output without any modifications. There are other algorithms such as Arithmetic Encoding, Punctured Elias Code, Golomb Code that have a higher compression ratio than RLE.

1. https://www.fileformat.info/mirror/egff/ch09_03.htm
2. <http://www.dspguide.com/ch27/2.htm>
3. Fredrick Heckland, 2004, Dept of Electronic and Telecommunication, NTNU, February 16 2004.
4. Niklas Wernersson, Source Channel Coding in Networks, KTH, Stockholm, 2008.
5. <http://effros.caltech.edu/jsc.html>