ARIZONA STATE UNIVERSITY

CSE 310, SLN 82170 — **Data Structures and Algorithms** — Fall 2020

Instructor: Dr. Violet R. Syrotiuk

Project #1

Milestone due Tuesday 09/15/2020; complete project due Thursday 10/01/2020

Encoding and decoding schemes are used in a wide variety of applications, such as in music or video streaming, data communications, storage systems (*e.g.*, on CDs, DVDs, RAID arrays), among many others. In a *fixed-length encoding* scheme each character is assigned a bit string of the same length. An example is the standard ASCII code. One way of getting an encoding scheme that yields a shorter bit string on the average is to assign shorter codewords to more frequent characters and longer ones to less frequent characters. Such a *variable-length encoding* scheme was used in the telegraph code invented by Samuel Morse. In that code, frequent letters such a `e` (·) and `a` (· -) are assigned short sequences of dots and dashes while infrequent letters such as `q` (- - · -) and `z` (- - ··) have longer ones.

In this project you will implement a variable-length encoding and decoding scheme, and run experiments to evaluate the effectiveness of the scheme, and the efficiency of your algorithms.

Note: This project **must** be completed **individually**. Your implementation **must** use C/C++ and ultimately your code **must** run on the Linux machine `general.asu.edu`.

You **may not** use any external libraries to implement any part of this project, aside from the standard libraries for I/O and string functions (`stdio.h`, `stdlib.h`, `string.h`, and their equivalents in C++). If you are in doubt about what you may use, ask me.

By convention, your program should exit with a return value of zero to signal that all is well; various non-zero values signal abnormal situations.

You **must** use a version control system as you develop your solution to this project, *e.g.*, GitHub or similar. Your code repository **must** be *private* to prevent anyone from plagiarizing your work.

The rest of this project description is organized as follows. §1 gives the requirements for Project #1 including a description of the encoding and decoding schemes to implement in §1.1 and §1.2, respectively. §2 gives the experiments to design to evaluate the effectiveness and efficiency of the encoding and decoding schemes. Finally, §3 describes the submission requirements for the milestone and full project deadlines.

1 Program Requirements for Project #1

1. Write a C/C++ program that implements the encoding scheme described in §1.1 on plain text. The program **must** be compiled into an executable named `encode` and it **must** take one command line parameter. The parameter is one of the keywords `insertion` or `quick`. In all cases, you **must** read input from `stdin`, allowing redirection from a text file. You **must** write the encoded plain text to `stdout`, allowing redirection to a text file. See §1.1 for detailed instructions.
2. Write a C/C++ program that implements the decoding scheme described in §1.2 on input in the format produced by encoding scheme as prescribed in §1.1. The program **must** be compiled into an executable named `decode`. In all cases, you **must** read input from `stdin` allowing redirection from an encoded text file. You **must** write the decoded input to `stdout`, allowing redirection to a text file. The decoded input should match the original input, *i.e.*, the input prior to encoding. See §1.2 for detailed instructions.
3. Design experiments to evaluate your programs as described in §2. A brief report with figures plotting the data you collect, and interpretation of your results is expected.

Sample text input will be provided on Canvas; use them to test the correctness of your programs. Scripts will be used to check the correctness of your program. Therefore, **absolutely no changes to these project requirements are permitted**.

1.1 The Encoding Algorithm

Given a normal text file (redirected from `stdin`), for each line of the file:

1. Transform the line into a form that is more amenable to compression. The transformation rearranges the characters in the input into many clusters of repeated characters, in a way that it is possible to recover the original input.
2. Output the compressed form of the transformed line.

To transform a line of text, treat the line as a string of length N. First, compute (and store) every cyclic shift of the string, shifting to the left by one character. Then sort the N cyclic shifts in lexicographic order according to the ASCII code. Note the ordering of characters in the ASCII code.

Table 1 shows the transformation step for the example string `Mississippi`, of length $N = 11$. Under **Original**, rows with index $0, \ldots, 10$ show the cyclic shift of the string to the left by as many characters, *e.g.*, at index 5 is the string shifted cyclically to the left by 5 characters. Under **Sorted**, all N cyclic shifts of the string are sorted in lexicographic order. (The `next` column is used in decoding, so it is discussed in §1.2.)

Table 1: Transformation Step of the Encoding Algorithm

Index	Original	Index	Sorted	next
0	M i s s i s s i p p i	0	M i s s i s s i p p i	4
1	i s s i s s i p p i M	1	i M i s s i s s i p p	0
2	s s i s s i p p i M i	2	i p p i M i s s i s s	6
3	s i s s i p p i M i s	3	i s s i p p i M i s s	9
4	i s s i p p i M i s s	4	i s s i s s i p p i M	10
5	s s i p p i M i s s i	5	p i M i s s i s s i p	1
6	s i p p i M i s s i s	6	p p i M i s s i s s i	5
7	i p p i M i s s i s s	7	s i p p i M i s s i s	2
8	p p i M i s s i s s i	8	s i s s i p p i M i s	3
9	p i M i s s i s s i p	9	s s i p p i M i s s i	7
10	i M i s s i s s i p p	10	s s i s s i p p i M i	8

The compressed output for a non-empty line consists of two lines:

1. The index of the row in which the original string appears in the **Sorted** column.
2. Form a string `last` consisting of the last character of each **Sorted** string. In this string, which is some permutation of the original string, characters form clusters of characters of size one or more. To encode `last`, step through the string from left to right processing the clusters: For each cluster, output the cluster size, followed by a blank character, followed by the character in the cluster.

For the example string, the original string appears at index position zero in the **Sorted** column. The last character of each **Sorted** string is a new string `last=ipssMpissii`. The first two characters are each in a cluster of size one. This is followed by a cluster of size two of the character `s`, and so on. Therefore, the encoding of the string `ipssMpissii` is:

```
0
1 i 1 p 2 s 1 M 1 p 1 i 2 s 2 i
```

1.1.1 Input to the Encoding Algorithm

The encoding algorithm has one input parameter taken from the command line: It is a keyword indicating the sorting algorithm to use. The text to encode **must be** read from `stdin`, which may be redirected from a file in Unix/Linux format. It is important to know that in Window files, the end of line is signified by two characters, Carriage Return (CR) followed by Line Feed (LF) while in Linux, only LF is used. Similarly, your output **must be** written to `stdout`, which may be redirected to a text file.

Some examples of input to your `encode` program:

```
encode insertion <ex1.txt >encoded_ex1.txt
encode quick <ex2.txt >encoded_ex2.txt
```

You must implement both the Insertion Sort algorithm and the Quicksort sorting algorithms. If the keyword is equal to `insertion` then you must use Insertion Sort to sort the strings in **Original**. Likewise, if the keyword is `quick` then the sorting algorithm used is Quicksort.

Do not sort the strings directly because it will result in too much data movement. To be efficient, sort pointers to the strings instead.

1.1.2 Output of the Encoding Algorithm

The output of your encoding scheme is text in which each line of the input is compressed as described in §1.1. Two lines of output are produced for every non-empty line of input. *If a line consists of a LF control character only, then the line is empty; the output is a LF character.*

1.2 The Decoding Algorithm

Now we describe how to decode a line, *i.e.*, recover the original line. There are three steps in the recovery:

1. Read the integer giving the `index` of the row in which the original string appears in the **Sorted** column.
2. Recover the string `last`.
3. Using the `index`, `last`, and knowledge of the `next` column, recover the original string.

Recall that the encoded output of the line with the string `Mississippi` is:

```
0
1 i 1 p 2 s 1 M 1 p 1 i 2 s 2 i
```

In this case, the `index` is zero. Recovering the string `last`=`ipssMpissii` from the encoded string is straightforward. Using `index`, `last`, and knowing the `next` column in Table 1, makes decoding easy, as given by the following pseudocode (which, of course, you must generalize):

```
int i, index, pos;
index = 0; // Index of row in which original string appears
int next[11] = { 4, 0, 6, 9, 10, 1, 5, 2, 3, 7, 8  }; // See section 1.2.1
char last[12] = "ipssMpissii"; // Recovered string last
pos = next[ index ];
for( i = 0; i < 11; i++ ){
   putchar( last[ pos ] );
   pos = next[ pos ];
}
```

This results in the output `Mississippi`, *i.e.*, the original string of the line is successfully recovered.

1.2.1 Computing the `next` Column

We now describe how the `next` column is computed from the **Sorted** strings. For $i = 0, \ldots N-1$, `next[`i`]` is the index of the row containing the cyclic shift of **Sorted**[i] to left by one character. For the example in Table 1, **Sorted**[**0**] is `Mississippi`. Shifting this string to left by one character gives `ississippiM`, and this string can be found at **Sorted**[**4**]. Hence `next[0]` is 4. **Sorted**[**1**] is `iMississipp`. Shifting this string to left by one character gives `Mississippi`, and this string can be found at **Sorted**[**0**]. Hence `next[1]` is 0. **Sorted**[**2**] is `ippiMississ`. Shifting this string to left by one character gives `ppiMississi`, and this string can be found at **Sorted**[**6**]. Hence `next[2]` is 6. Continuing in this way for $i = 3 \ldots 10$ gives the values in the column `next` in the table. Indeed, `next` is just a permutation of the indices $0, \ldots, N-1$.

What is amazing is that the information in the encoding is enough to reconstruct `next`, and therefore the original message! From `last`, we know all of the characters in the original string, they're just permuted. We can reconstruct the first column in **Sorted** by sorting the characters in `last`; see Table 2.

Because `M` only occurs once in the string and the array is formed using cyclic shifts, we can deduce that `next`[0] = 4 because `M` is in the last column of row with index 4. However all the other characters are in clusters of size larger than one, so how can we tell how to compute `next`? For character `p`, it may seem ambiguous whether `next`[5] = 1 and `next`[6] = 5, or whether `next`[5] = 5 and `next`[6] = 1.

Table 2: Reconstructing `next` from the Encoding

Index	**Sorted**	`next`
0	`M ? ? ? ? ? ? ? ? ? i`	4
1	`i ? ? ? ? ? ? ? ? ? p`	
2	`i ? ? ? ? ? ? ? ? ? s`	
3	`i ? ? ? ? ? ? ? ? ? s`	
4	`i ? ? ? ? ? ? ? ? ? M`	
5	`p ? ? ? ? ? ? ? ? ? p`	1
6	`p ? ? ? ? ? ? ? ? ? i`	5
7	`s ? ? ? ? ? ? ? ? ? s`	
8	`s ? ? ? ? ? ? ? ? ? s`	
9	`s ? ? ? ? ? ? ? ? ? i`	
10	`s ? ? ? ? ? ? ? ? ? i`	

As it turns out, there is a rule that resolves the ambiguity. It is: If row index i and j both start with the same letter and $i < j$, then `next`$[i]$ < `next`$[j]$. This rule implies that `next`[5] = 1 and `next`[6] = 5.

Why is this rule valid? The rows are sorted, so row 5 is lexicographically less than row 6. This means that the nine unknown characters in row 5 must be less than the nine unknown characters in row 6 (since both rows start with the letter `p`). We also know that between the two rows that end with `p`, row 1 is less than row 5. But, the nine unknown characters in row 5 and 6 are precisely the first nine characters in rows 1 and 5. Thus, `next`[5] = 1 and `next`[6] = 5 or this would contradict the fact that the strings are sorted.

Using the rule allows all remaining ambiguities to be resolved and all entries of `next` to be computed.

1.2.2 Input to the Decoding Algorithm

The input to the decoding scheme is text in the form generated by the encoding scheme. As with the encoding scheme, the text to decode **must** be read from `stdin`, which may be redirected from a file in Linux format. Similarly, your output **must** be written to `stdout`, which can be redirected to a text file.

If the input to the encoding scheme consists of k non-empty lines, then its output has $2k$ encoded lines. Thus the decoding scheme iterates k times in order to decode the k encoded non-empty lines of input.

1.2.3 Output of the Decoding Algorithm

The output of the decoding scheme should equal the input to the encoding scheme. Note that some care will be needed to take care of the LF characters so that the lines match. (Try using the `diff` commands to compare the two files.)

2 Experimentation

A standard measure of the "goodness" of a compression algorithm's effectiveness is the *compression ratio*. This is the ratio $\frac{t-c}{t} \times 100\%$, where t is the total number of characters in the input, and c is the number of clusters in the encoding.

For example, the encoding of `Mississippi` is `1 i 1 p 2 s 1 M 1 p 1 i 2 s 2 i`. In this example, the total number of characters is $t = 11$, the number of clusters is $c = 8$, and so the compression ratio is $\frac{3}{11} \times 100$ or 27%. (Here, we are ignoring the fact we used integers to code the cluster sizes; this can be done more intelligently but this project is already enough work, right? ☺)

Design a set of experiments to study:

1. The average compression ratio; in addition to the average, compute the minimum, maximum, and standard deviation of the compression ratio. You might consider using a box and whiskers plot for this metric.
2. The time to encode each input for each type of sort, *i.e.*, for Insertion and for Quicksort. Plot the run time as a function of input size.
3. The time to decode each encoded input. Plot the run time as a function of input size.
4. The compression ratio as a function of number of lines encoded. The encoding algorithm has been described as encoding one line at a time. If you instead encode $2, 3, \ldots$ lines at a time, does the compression ratio improve? What do you expect to happen?

3 Submission Instructions

Submissions are always due before 11:59pm on the deadline date. Do not expect the clock on your machine to be synchronized with the one on Canvas!

1. The milestone is due on Tuesday, 09/15/2020. See §3.1 for requirements.
2. The complete project is due on Thursday, 10/01/2020. See §3.2 for requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted.

An unlimited number of submissions are allowed. The last submission will be graded.

3.1 Requirements for Milestone Deadline

For the milestone deadline you must implement the encoding scheme described in §1.1 and only support the `insertion` keyword. Using the submission link on Canvas for the Project #1 milestone, submit a zip[1] file named `yourFirstName-yourLastName.zip` that unzips into the following:

Project State (5%): In a folder (directory) named `State` provide a brief report (.pdf preferred) that addresses the following:

1. Describe any problems encountered in your implementation for this project milestone.
2. Describe any known bugs and/or incomplete implementation in the project milestone.
3. While this project is to be completed individually, describe any significant interactions with anyone (peers or otherwise) that may have occurred.
4. Cite any external books, and/or websites used or referenced.

Implementation (25%): In a folder (directory) named `Code` provide:

1. In one or more files, your well documented C/C++ source code implementing the encoding scheme required for this project milestone.
2. A `makefile` that compiles your program and produces an executable named `encode` on `general.asu.edu`. Our TA will write a script to compile and run all student submissions on `general.asu.edu`; therefore executing the command `make encode` in the `Code` directory must produce the executable `encode` also located in the `Code` directory.

[1] **Do not** use any other archiving program except `zip`.

Correctness (70%): The correctness of your program will be evaluated by running a number of tests on a text files, some of which will be provided to you on Canvas prior to the deadline for testing purposes. For the milestone deadline, the script will only test your `encode` program. As stated several times, your program **must** read input from standard input. **Do not use file operations to read the input!**

The milestone is worth 30% of the total project grade.

3.2 Requirements for Complete Project Deadline

For the full project deadline, you must implement both the encoding and decoding schemes, both sorting algorithms, as well as conduct experiments with each program summarized in a report. Using the submission link on Canvas for the complete Project #1, submit a zip[2] file named `yourFirstName-yourLastName.zip` that unzips into the following:

Project State (5%): Follow the same instructions for Project State as in §3.1.

Experimentation and Report (15%): In a folder (directory) named `Report` provide a brief report (.pdf preferred) that addresses the following:

1. Describe the experiments you ran, *i.e.*, the characteristics of the input data you used, such the input sizes in lines and characters, among others.
2. Present figures/tables plotting the results of your experimentation as requested in §2. Use the data you collected to interpret your results. Can you draw any general conclusions about the compression ratio, about the impact of the sorting algorithm on the run time of the encoding scheme, about the impact of input size on the decoding scheme, about the impact on the compression ratio as a function of the number of lines encoded?

Implementation (20%): Follow the same instructions for Implementation as in §3.1, except that the TA should be able to `make` both the `encode` and `decode` executables on `general.asu.edu` in your `Code` directory.

Correctness (60%): The same instructions for Correctness as in §3.1 apply except that the input will test both the encoding and decoding schemes.

4 Marking Guide

The project milestone is out of 100 marks.

Project State (5%): Summary of project state, use of a zip file, and directory structure required (*i.e.*, a folder/directory named `State` and `Code` is provided).

Implementation (25%): 15% for the quality of implementation in your encoding scheme; 5% for reading from `stdin` and writing to `stdout`; 5% for a working `makefile`.

Correctness (70%): For correct output on at least 7 sets of sample input.

The full project is out of 100 marks.

Project State (5%): Summary of project state, use of a zip file, and directory structure required (*i.e.*, a folder/directory named `State`, `Report`, and `Code` is provided).

Experimentation and Report (15%): Experiment design, results (plots/tables) of results gathered, and interpretation of results.

Implementation (20%): 15% for the quality of implementation in your code; 5% for reading from `stdin` and writing to `stdout`, and for a working `makefile`.

Correctness (60%): 60% for correct output on at least 10 sets of sample input.

Comments will be provided to you when your graded project is returned.

[2] **Do not** use any other archiving program except `zip`.