ARIZONA STATE UNIVERSITY

CSE 310, SLN 30912 — **Data Structures and Algorithms** — Spring 2022

Instructor: Dr. Violet R. Syrotiuk

Project #1

Milestone due Sunday, 02/06/2022; complete project due Sunday, 02/20/2022

Encoding and decoding schemes are used in a wide variety of applications, such as in the streaming of music and videos, data communications, storage systems (*e.g.*, on CDs, DVDs, RAID arrays), among many others. In a *fixed-length encoding* scheme each symbol is assigned a bit string of the same length. An example is the standard ASCII code which uses 7 bits for each symbol. One way of getting an encoding scheme that yields a shorter bit string on average is to assign shorter codewords to more frequent symbols and longer ones to less frequent symbols. Such a *variable-length encoding* scheme was used in the telegraph code invented by Samuel Morse. In Morse code, frequent letters such as `e` (·) and `a` (· -) are assigned short sequences of dots and dashes while infrequent letters such as `q` (- - · -) and `z` (- - ··) are assigned 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**. You **must** write your solution in C/C++. Your code will be evaluated in Gradescope (Ubuntu 18.04 operating system, with version 7.5.0 of `gcc` and `g++`).

You **may not** use any external libraries to implement any part of this project, aside from the standard libraries for I/O, dynamic memory allocation, and string functions (*i.e.*, `stdlib.h`, `stdio.h`, `string.h`, and their equivalents in C++). In particular, you **may not** use the `vector` class. If you are in doubt about what you may use, ask on `Ed Discussion`.

On the other hand, you **must** use the definitions of constants and structures in the `defns.h` file provided. No error checking of input is required; you may assume the input is in the described format.

By convention, any program you write must exit with a **return value of zero** to signal normal completion; you may introduce non-zero return values to signal abnormal completion.

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 the preprocessing required in §1.1, followed by a description of the encoding and decoding algorithms in §1.2 and §1.3, 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. **No deviations to these project requirements are permitted**.

1 Program Requirements for Project #1

1.1 The Preprocessing Algorithm

Write a C/C++ program that performs preprocessing on plain ASCII text to be used in encoding and decoding. Use a `makefile` to compile your program into an executable named `preprocess`. The text to process **must be** read from `stdin`, which may be redirected from a file, is assumed to be in Linux format.[1] Your output **must be** written to `stdout`, which may be redirected to a text file.

An example of an invocation of your `preprocess` program:

```
preprocess <file.txt >pre-file.txt
```

[1] 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.

While the preprocessing algorithm is very simple, the encoder and decoder are not. The major steps of the preprocessing algorithm are:.

1. In the file `defns.h` is the definition of a structure named `symbol`. Allocate an array named `Symbols` of `struct symbol` of size `NSYMBOLS`, the number of symbols in the ASCII code. Initialize the `symbol` field to the ASCII character corresponding to its decimal value, its `freq` to zero, all the pointer fields to `NULL`, and the `encoding` to the empty string; see ASCII code. *This array can and should be indexed by the decimal value corresponding to a symbol.*

2. Read the input, character by character, from `stdin`. For each character read, increment the `freq` field of its corresponding entry in the array `Symbols`. At the end of this step, you will know the number of occurrences of each ASCII symbol in the input.

3. Now iterate through the array `Symbols`. For each index position of non-zero frequency, write the `index` position, followed by a tab, and the `freq` field, followed by a newline (`\n`) to `stdout`.

The output of the preprocessing algorithm is specific to a set of input.

1.2 The Encoding Algorithm

Write a C/C++ program that produces an encoding of plain ASCII text, utilizing the output of the preprocessing phase on the same text. Use a `makefile` to compile your program into an executable named `encode`. Your encoder program **must** read two command line parameters: (1) the name of a text file, and (2) a keyword that is either `insertion` or `merge`. The text to encode **must be** read from `stdin`, which may be redirected from a file assumed to be in Linux format. Assume that the file named as the first parameter contains the output of the preprocessing algorithm run on the text being encoded.

An example of an invocation of your `encode` program, consists of a sequence:

```
preprocess <file.txt >pre-file.txt
encode pre-file.txt insertion <file.txt >encoded-file.txt
```

Encoding proceeds in two steps: (1) compute the encoding for each symbol in the input; (2) use the symbol encoding to encode the input. These steps are described in the next two subsections.

1.2.1 Compute the Encoding for Each Symbol

A video is provided that works through a detailed example of the steps to compute the encoding for each symbol. Work through the steps with the example to be sure you understand them. Do not start your implementation until you understand these steps. This is the most complex part of this project.

The major steps to compute the encoding for each symbol in the input follow:

1. Follow Step 1 of the preprocessing algorithm in §1.1 to initialize an array named `Symbols` of `struct symbol` of size `NSYMBOLS`. Read the contents of the file named in the first command line parameter and use it to update the `freq` field for each symbol.

2. Create two new arrays of `struct tree`, one to be used only for alphabetic symbols named `Alpha`, *i.e.*, for the symbols `A-Z` and `a-z` only, and a second named `NonAlpha` for all other symbols, occurring in the input. Initialize the `index` to the decimal value of the symbol. Copy the `symbol` and `freq` fields from the fields of the same name in `Symbols[index]`. Finally, initialize the `root` field to `Symbols[index]`[2].

3. Sort each array in increasing order by `freq` field, using the sorting method specified in the second command line parameter, *i.e.*, if the keyword is `insertion` then use Insertion Sort for sorting; if the keyword is `merge`, then use Mergesort for sorting. Break ties by the lexicographic ordering of the characters, *i.e.*, the decimal value corresponding to the character; again, see the ASCII code.

[2]The address of an array is its name, *e.g.*, `Symbols`. The address of an array entry it its index position, *e.g.*, `Symbols[index]`.

4. Initially the array `Alpha` can be considered to be n one-node trees ordered by `freq`, where n is the total number of alphabetic symbols occurring in the input. Specifically, each tree consists of a single leaf node labelled by `symbol`. Repeat the following steps until a single binary tree is obtained.

   (a) Take the two trees, t_L and t_R, of smallest frequency, *i.e.*, `Alpha[0]` and `Alpha[1]`.
   (b) Allocate a new node t of type `struct symbol`; set its `parent` field to `NULL`.
   (c) Set the left subtree of t to the `root` field of t_L, and the right subtree of t to the `root` field of t_R.
   (d) Set the `freq` field of t to the sum of the `freq` of t_L and t_R.
   (e) Delete t_L and t_R from `Alpha`, *i.e.*, delete `Alpha[0]` and `Alpha[1]`.
   (f) Insert t into the array `Alpha` according to its `freq` field. If there are ties, insert t to the right of all other trees with equal frequency. Suppose the position of insertion in `Alpha` is `p`. The `index` and `symbols` fields are not used (except by leaf nodes). Set the `freq` field in `Alpha[p]` to the `freq` field in t. Set the `root` field in `Alpha[p]` to point to t.

   At the end of this step, a single binary tree will have been constructed bottom-up containing all alphabetic characters in the input as leaf nodes. The `freq` field of the root node of this tree should equal the total number of alphabetic symbols in the input.

5. Repeat the steps of the construction of a binary tree in step 4, this time on the array `NonAlpha` of all non-alphabetic characters. At the end of this step, a single binary tree will have been constructed bottom-up containing all non-alphabetic characters in the input as leaf nodes. The `freq` field of the root node of this tree should equal the total number of non-alphabetic symbols in the input.

6. Now allocate a new node `Root` of type `struct symbol` and form a single binary tree with the binary trees constructed in steps 4 and 5, as left and right subtrees, respectively.

7. To determine the encoding for each symbol in the array `Symbols`, follow successive `parent` pointers up to the `Root` of the tree constructed in step 6, concatenating edge labels as you work your way up the tree. Left edges are labelled by zero and right edges by one. The encoding of the symbol is the reverse the binary string obtained.

8. Write the total number of symbols in the input with frequency greater than zero, followed by a newline (`\n`) to `stdout`. Then, iterate through the `Symbols` array. For each symbol whose frequency is greater than zero, write the `index` position, followed by a tab, the `symbol` field, followed by a tab, and the `encoding` field, followed by a newline (`\n`) to `stdout`.

1.2.2 Use the Encoding of Symbols to Encode the Input

Once the encoding of each symbol has been computed, encoding the input is straightforward. For each line of input[3] one line of output is produced. Initialize the encoding of a line to an empty string. For each character in the line:

1. Look up the encoding of the symbol in `Symbols` and append it to the encoding of the line, including the encoding of the a newline (`\n`) character.

2. Write the encoded line, followed by a newline (`\n`) to `stdout`.

If a line consists of a newline character only, this produces an encoding of the newline symbol, followed by a newline (`\n`).

Each encoded line is simply a bit string; it **must not** have any blank characters between encoded symbols.

[3] A line is terminated by a newline (`\n`) character.

1.3 The Decoding Algorithm

Write a C/C++ program that decodes encoded text into ASCII text, utilizing the output of the preprocessing phase on the same text. Use a `makefile` to compile your program into an executable named `decode`. Your encoder program **must** read two command line parameters: (1) the name of the text file, and (2) a keyword that is either `insertion` or `merge`. The text to decode **must be** read from `stdin`, which may be redirected from a file assumed to be in Linux format. Assume that the file named as the first parameter contains the output of the preprocessing algorithm run on the text being decoded.

An example of an invocation of your `decode` program, consists of a sequence:

```
preprocess <file.txt >pre-file.txt
decode pre-file.txt merge <encoded-file.txt >decoded-file.txt
```

After decoding, the files `file.txt` and `decoded-flie.txt` should be identical.

1.3.1 The Decoding Algorithm

Decoding proceeds in two steps: (1) rebuild the tree used in encoding; (2) use the tree to decode the encoded input. In more detail:

1. Follow steps 1-6 in §1.2.1 to reconstruct the binary tree rooted at `Root`, using the sorting algorithm specified by the second command line parameter in step 3.
2. Read n, the total number of symbols in the input with frequency greater than zero. Skip over the next $n + 1$ lines summarizing the encoding produced.
3. For each line of the input, initialize the decoding of a line to an empty string. Repeat until the newline character:
   (a) Starting at the root, if the current bit is zero move into the left subtree, otherwise move into the right subtree. Advance to the next bit.
   (b) If a leaf node is reached, append the `symbol` to the decoding of the line. Return to the `Root` of the tree to decode the next symbol.
4. Once the newline is decoded write the decoded line, including the newline to `stdout`.

2 Experimentation

A standard measure of the "goodness" of an encoding and decoding algorithm's effectiveness is the *compression ratio*. Let n be the total number of symbols in the input. Had we used a fixed-length encoding for the input, we would have to use at least $\lceil \log_2 n \rceil$ bits for each symbol. Using the `freq` field, and the length of the `encoding` field in bits, we can compute the expected number of bits per symbol in our encoding. For each symbol of non-zero occurrence probability:

$$\text{ExpectedNumberBits} = \sum \frac{\texttt{freq}}{n} \times \text{LengthInBits}(\texttt{encoding}).$$

The compression ratio is then:

$$\text{CompressionRatio} = \frac{\lceil \log_2 n \rceil - \text{ExpectedNumberBits}}{\lceil \log_2 n \rceil} \times 100\%.$$

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 sort input instances for each type of sort, *i.e.*, for Insertion Sort and for Mergesort. Plot the run time as a function of input size on one figure. For this experiment you should run your sorting algorithms outside of the context of the encoding/decoding problem. Do the running times cross each other? Which algorithm is better for sorting smaller instances? Which algorithm is better for sorting larger instances?

3. The time to encode and decode input instances. Plot the run time as a function of input size.

4. How do you expect the compression ratio to vary according to n?

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/Gradescope!

1. The milestone is due on Sunday, 02/06/2022. See §3.1 for requirements.
2. The complete project is due on Sunday, 02/20/2022. 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 are to implement the preprocessing algorithm described in §1.1, and the encoding algorithm described in §1.2. You only need to support the `insertion` command line parameter, *i.e.*, *all sorting is to be done using the Insertion Sort algorithm* for the milestone deadline. (You still need to read the parameter on the command line.)

Submission instructions for the milestone as well as a grading rubric will be posted on Canvas. A demo of Gradescope will be provided in recitations. Sample input will be provided on Canvas.

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

3.2 Requirements for Complete Project Deadline

For the full project deadline, you are to implement all three algorithms, the preprocessing algorithm, and both the encoding and decoding algorithms, including both the Insertion Sort and Mergesort sorting algorithms. You are also required to conduct the experiments with both `encode` and `decode` described in §2, and summarize your results, and interpretation of those results in a report. The report should be in PDF format.

Submission instructions for the full project, as well as a grading rubric will be posted on Canvas.