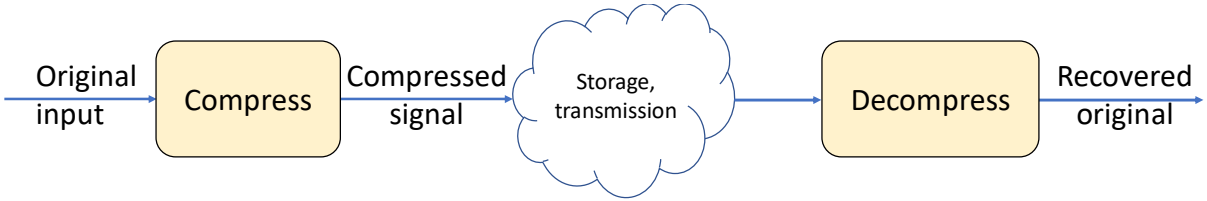
CS112 – Spring 2023

Project02 – Text Compression via Huffman Coding

INTRODUCTION

For **Project02** you will develop programs that compress and decompress text files. Data compression is a critical technology—it enables significant cost and resource savings for many data storage and transmission applications:

- Storage of audio and video on DVDs, Blu-rays, etc
- Audio and video streaming, e.g. by Netflix, Spotify, etc
- Audio compression for cellular telephony, digital radio, etc
- Computer data storage and transmission



There are two general categories of compression technologies:

- Lossless – the decompressed results are identical to the original data before compression
- Lossy – the decompressed results are very similar to but are not identical to the original

Why would anyone ever use lossy compression? Lossless techniques can achieve “compression ratios” (size of original data / size of compressed data) of 2:1 or maybe 3:1. Today it is common for High Definition video to be compressed with lossy compression almost 1000:1.

In this project, we will learn about one basic lossless compression technology called "Huffman encoding". We will use that technology to develop programs for lossless compression and decompression of text files.

Morse Coding

Lossless compression methods all rely on the fact that some **input symbols** (text characters in our case) occur more often than others. For example in English text, the letter ‘e’ occurs much more often than the letter ‘q’. If we use fewer bits to transmit a compressed ‘e’ than we use for a compressed ‘q’, then since we send the letter ‘e’ so often, we should be able to compress text compared to the original. Of course, the

compressed symbol for every letter of the alphabet should be chosen to reflect the letters' probabilities: long compressed symbols for less common letters, and short compressed symbols for frequently occurring letters.[1]

Probably the most famous compression technique for English text is Morse Code, developed by inventors Alfred Vail and Samuel F. B. Morse for use with the telegraph.[2] Morse Code encodes every Latin-alphabet letter and every numeral from 0 to 9 with a series of brief "dots" and longer "dashes". These can be transmitted electrically as two different voltages, as sounds, light flashes, etc. In modern Morse Code, the compressed symbol for the letter 'e' is a single dot "·" and the compressed symbol for 'q' is "- - · -".

In Morse Code, the compressed symbol for the letter 's' is "· · ·". Do you see the problem with this?

[1] Ideally, the length of the coded symbol for each letter should be $-\log_2$(Probability of the letter), so if the probability is 0.5, the length is 1. If probability = 0.25, length = 2, etc.

[2] https://en.wikipedia.org/wiki/Morse_code, downloaded July 20, 2022

A Problem with Morse Code

If the compressed symbol for ‘s’ is “· · ·” and the compressed symbol for ‘e’ is “·”, then how can a receiver distinguish an ‘s’ from the pattern ‘eee’ ? In fact, Morse Code uses three values to build its **compressed alphabet** (the set of all **compressed symbols**):

- Dot
- Dash
- silence

The duration of silence between the dots and dashes is significant: the three dots in the symbol for ‘s’ occur closer together than the three dots in the symbol for ‘eee’. This is inefficient and is certainly difficult for computers, which work natively in binary. But it works well for human Morse code experts, who like a space between letters, a longer space between words, and a longer space between sentences.

Huffman Coding

Huffman Coding is similar in concept to Morse Code, but Huffman Coding truly uses only two values to build its compressed alphabet, which are usually denoted as ‘0’ and ‘1’. The key to the design of Huffman Coding is that the compressed symbol for each input symbol is not a prefix for the compressed symbol of any other input symbol. Once we have seen some pattern of ‘0’s and ‘1’s that matches the compressed symbol for a particular input, we do not have to guess whether the pattern indicates our input or is the beginning of the symbol for some other input. Here is an example of a Huffman Code for five possible inputs that shows this property:

Input	Compressed Symbol
α	00
β	01
γ	100
δ	101
ε	11

Let’s verify this “no-prefix” property…

- If the first value you receive is a ‘0’ then the next received value tells you whether the input was ‘α’ or ‘β’.
- If you first receive a ‘1’ then wait for the next received value
    - If it is a ‘1’ then the input was an ‘ε’
    - If it is a ‘0’ then the third received value determines whether the input was ‘γ’ or ‘δ’

- The next received value must be the start of the compressed symbol for the following input character

Construction of Huffman Codes

There is a clever algorithm for the construction of Huffman codes that relies on trees and lists—if you finish Part 1 and Part 2 of this project early and want to learn about it, see the instructor and he will give you a description of how to code it in Java.

But for this project, you will be provided a file named **codebook** (in the **CourseInfo/Project02** directory) that contains the Huffman code that will use.

The codebook File

The **codebook** file consists of multiple lines, one line for each character value that your Huffman coder must handle. Each line shall have the following format:

<the character's UTF value as a decimal integer><colon><compressed symbol for the character as text '0's and '1's><newline>

For example:

61:1100111

For this project the Huffman coder shall support only the following input character values:

- '\u0004': the EOT "End Of Transmission" character. The encoder shall send this after processing all its input, to signal to the decoder that the input has ended
- '\u0007' through '\u00fe' inclusive. This is a small subset of UTF-16 but is larger than the old ASCII character set.

The **codebook** file has one line for each of these character values.

The Project

For this project, you will write a Huffman encoder that takes a text file and encodes it to a pattern of '0's and '1's using a Huffman code. And you will write a Huffman decoder that decodes a file of '0's and '1's to reconstruct the original text.

Part 1 – Testing

The programs are not too difficult to write, but there are a lot of tricky details to evaluate. So for Part 1 of the project, you will not work on the programs—you will work on a tester! This will be a Java program called **HuffTest.java** .

HuffTest.java takes two files as input: a text "input file" and the supposed output of your decoder. The file names are given as the first two command line arguments to **HuffTest.java**.

The tester shall read and compare the files to make sure they are identical. However, there is one detail to account for:

- The actual input file may contain characters that are not in the allowed set of characters described above in the "codebook file" section.
- If the encoder sees any disallowed characters, it must skip over them
- So your **HuffTest** also must skip any disallowed characters

If the input files to **HuffTest.java** match (ignoring disallowed characters), the program shall print "PASS" to `System.out`. If the files do not match, then **HuffTest** shall print to `System.out`:

FAIL input <<char from first file>> @ <<byte position in first file>> output <<char from second file>> @ <<byte position in second file>>

For example

FAIL input Z @ 541 output % @ 539

Of course, you must test your tester! As part of your project, you must turn in a brief informal document called **notes.txt**. In this document in the "Part 1" section, you will describe how you tested your tester. What kinds of good inputs did you test with? What kinds of bad inputs? Give some examples of your test data.

Phase 2 - Encoding and Decoding

Huffman encoding is pretty easy. For Part 2 of this project, you will write a Java program called **Encode.java**. This program will read in the **codebook** file and will store an entry for each character in the codebook—i.e. for each line in the **codebook** file—into a `HashMap`. The program takes two command-line arguments containing the name of the input text file and the output file of Huffman-coded data. Then, to encode:

- Read a character from the input file. Look up the value's compressed symbol in the `HashMap` and output the compressed symbol as a sequence of text '1's and '0's. The output should not have any spaces or newlines, just '0's and '1's
- Move on to the next input character and repeat
- After the last input character has been encoded, **Encode.java** must output the Huffman code for the "EOT" character. Then the program can close files and exit.

Also for Part 2, you will write a Java program called **Decode.java** that performs Huffman decoding. **Decode.java** also takes two command-line arguments: an input file containing Huffman-coded data and an output file containing decoded text.

Decoding requires translating the `HashMap` of the data in the **codebook** file into a tree:

- You will need to make a Node class that stores a character, its Huffman symbol, and links to left and right neighbors
- You will make a tree of these Nodes. Only the "leaf" Nodes store actual characters. The root Node and other internal Nodes just store connections among Nodes. You start by creating an empty root Node.
- You will read each character's information from the **codebook** file. For each character:
    - Look at the character's Huffman symbol, which is a list of '0's and '1's. Start with a pointer to the root of your Huffman tree. For each '0' character, move your pointer one step down the left branch of the current Node. Make a new Node if there is not one there already. For each '1' character, move your pointer one step down the right branch of the current Node, making a new Node if needed.
    - When you have walked down the tree properly for each '0' and '1' in the current character's Huffman symbol, then store the character value and its Huffman symbol in the current Node.
    - Since all characters in codebook have a unique Huffman symbol, each character will have its own unique Node in your Huffman tree!

With this Huffman tree, decoding each character from the input is also pretty simple:

- Have a Node pointer point to the top of the Huffman tree, and read a "0" or "1" from the compressed input
- If the value read is "0", then change your pointer so it points to the left-side child of the pointed-to Node
- If the bit read is "1", then point to the right-side child of the pointed-to Node
- If the resulting node is a “leaf Node”, output the corresponding input character, and then start decoding the next character
- If the resulting node is still not a “leaf Node”, then read another value from the compressed input and traverse further down the tree

As you decode all the '0's and '1's from the input file, eventually you will decode the EOT character. Do not output this character, but when you decode it, you can close all files and exit **Decode.java**.

With your **HuffTest** program, you should be able to test your **Encode.java** and **Decode.java** pretty easily. Please add a "Part 2" section to your **notes.txt** file describing how you tested your encoder and decoder. Please explain not just the test method, but also how you picked your test data. Please include an example or two.

Your Programs

This project consists of three programs and one file:

- **HuffTest** – test output files to be sure they match the input
- **Encode** - Compress text files, given the Huffman table in **codebook**
- **Decode** – Decompress compressed files, hopefully yielding the original text files
- **notes.txt** – your testing document

Grading

The main goal is for your decoder output to match your encoder input exactly (ignoring disallowed characters in the encoder).

If you look at the size of your input file and your compressed file, why is the compressed file so big? It stores '0' and '1' values as text, using 8 bits (probably) for each character. In an actual data compression or transmission application, we would handle the '0's and '1's as individual bits, and your compressed file would be 1/8 as large. (Extra credit??? Just kidding—no.)

Put your three programs and **notes.txt** into a subdirectory called **Project02** inside your **MyWork** directory, and remember to push your **Project02** to GitHub before the deadlines.

- Part 1 (**HuffTest.java** and first draft of **notes.txt**) is due before 11:59pm on May 2nd
- Part 2 (**Encode.java**, **Decode.java** and final draft of **notes.txt**) is due before 11:59pm on May 10th
- **I will not accept any submissions after 11:59pm on Friday May 12th**, the end of the semester…and you need late-day-credits if you want to use the extra two days.

Rubric

Milestone	Points	Comments
Part 1 - **HuffTest** compiles successfully and correctly reports on various sets of test files	45	9 points each for 5 test cases
Part 1 – **notes.txt**	5	

Part 2 - **Encode** compresses input textfiles as much as my reference implementation. And **Decode** produces output that matches the original input	70	10 points each for 7 test cases
Part 2 – software quality of **Encode.java** and **Decode.java**	20	Judged subjectively by graders
Part 2 – **notes.txt** includes accurate and useful descriptions of how all three programs were tested	30	Judged subjectively by graders

Conclusion

This project ties together many of the techniques you have practiced during the course:

- File processing
- Text manipulation
- Advanced data structures
- Recursion
- Excellent test planning and execution

Congratulations on a job well done!