

University of Pennsylvania
Department of Electrical and Systems Engineering
Digital Audio Basics

ESE250 Spring 2012

Lab 3: Loss Compression

Tuesday, January 31, 2012

For Lab Session: Tuesday, January 31, 2012 in Detkin Lab.

Due: Monday, February 6, 2012 by 12:00pm - noon.

Collaboration: Work in lab in teams of 2 (we will pair you up). Perform individual writeups. See course collaboration policy in the [Administrative Handout](#).

Objective: Re-encode files to reduce space; quantify savings.

Prelab Requirements:

- Read this entire lab assignment, especially the Motivation section, the Encoder subsection of the Lab Procedure, and the Theory subsection of the Lab Report Guidelines section.
- Download the zip file online with the Java files and explore them with the help of the descriptions given in the lab manual. (You can open these files in notepad++ if you do not have eclipse installed). In Encoder.java, you will find comments that tell you to insert code. Write pseudocode for your proposed implementation of Encoder.java, where the comments indicate. If you do not know what pseudocode is, refer to the wikipedia page: [Pseudocode](#). You do not have to write the actual code.
- Answer the Prelab questions at the bottom of this document.

Deliverable:

- Names of all lab group members
- Answers to all lab questions (including prelab question)
- GenerateCode.java
- All dictionary files created
- All the compressed files produced by all the file-dictionary pairs from the table on the last page.

Handin: All labs will be turned in electronically through the [Penn Blackboard](#) website. Go to the assignment submission link and follow the instructions. Please submit all your files as **one zip file**. Also, remember that your writeup should be a **PDF file**.

Algorithm Checkpoint: Explain to your TA, how your pre-lab `encode` pseudocode works.

Exit Ticket: Demonstrate a dictionary, encoding, compression rate, decoding and byte comparison of one of the encoded files.

Motivation

The way we perceive information is not necessarily the most compact way of representing that information. Frequency-based encoding recognizes this and tries to compress the representation by assigning shorter representations to more frequent symbols. In this lab you will generate the encoding assignments to most efficiently compress text files, encode multiple files using different encodings and report on the compression rates achieved.

Lab Procedure

For this lab, you will edit the Java file `GenerateCode.java` so that it reads a file and counts the frequency of the characters within that file. Although you are free to use any code editor that you want, we highly recommend that you take this opportunity to learn how to use Eclipse, an advanced integrated development environment (IDE) that you can use to code in Java. If you wish to use Eclipse, download [lab3.zip](#) (do not extract it) and follow the instructions at the end of the lab in the section titled Eclipse. If you decide to use a different editor, you should instead download [Encoder.java](#), [Decoder.java](#), [CodeWord.java](#), [GenerateCode.java](#), [CodeWordDecoder.java](#), and set up your editor so that all five files are part of the `coding` Java package.

Encoder

An encoder takes a data file and a dictionary (that assigns each symbol in the data file to a binary encoding), and encodes the data file with the dictionary. For this lab, our symbols will consist of bytes (8 bits). A dictionary will map each of the 2^8 possible bytes to a binary encoding.

`Encoder.java` has one constructor, one field, `code`, and three methods, `main`, `encode` and `loadCode`. `code` is where the dictionary is stored once the dictionary file is read. It is an array of 256 Strings (composed of purely 0s and 1s), representing binary encodings, that is indexed by the byte that corresponds to each code. The constructor along with `loadCode` take care of populating this field.

Once the `code` array is set up, `encode` is called. It will read one byte at a time from the data file and place it in the integer `nextByte`. It will loop until it reaches the end of file, all the while keeping track of how many bytes it read. Finally, it will report how many bytes were read.

`Encoder.java` takes 3 command line arguments `<text file> <dictionary file> <output file>`¹ that tell the encoder on which files to operate and where to write its output.

¹The chevron brackets around each parameter, indicate that the arguments are variables. The chevrons should not appear in the actual command line

Building Bytes

Note that simply writing to a file the String that corresponds to the read byte, will not compress the data file but in fact will create an encoded file that is significantly larger than the original data file. This is due to the fact that each character in the string will itself be a byte and not a bit. What the Encoder does instead instead, is “build” bytes out of the strings it reads by grouping every 8 ones and zeros read and converting them into a byte.

Example

Consider the following dictionary:

Byte Number	Encoding
A	0001
B	10110
C	010
D	11

Suppose our data file is composed of the following bytes: `ABDCADCDC`. This should be encoded as follows:

Input	A	B	D	C	A	D	C	D	C
Encode	0 0 0 1	1 0 1 1 0	1 1	0 1 0	0 0 0 1	1 1	0 1 0	1 1	0 1 0
Output Bytes	27		104		117		10		

The last byte is not yet a full byte. The Encoder does more to it to ensure that it is a full byte before it writes it to the output file. Take note of what this is from the `Encoder.java` file and compare it to what you came up with for your pre-lab.

Iterators

In the provided implementation, `Encoder.java` stores the dictionary in an array of `Strings`. It iterates over each character in the string as it builds bytes, using an iterator. The skeleton of an iterator is shown below:

```
...
String encoding = ...
...
for (char nextCharacter : encoding.toCharArray()) {
    //Do something with nextCharacter
...

```

```
}
```

This construct will iterate over each character in the String `encoding`, putting the current character in `nextCharacter`. It starts with the left most character of the string and iterates until it reaches the right most.

Algorithm Checkpoint: Compare your pseudocode implementation to the provided implementation of `Encoder.java`. What are the main differences? Is one more efficient? Why or why not?

To do: Your job is to determine the most efficient encoding to compress various text documents. To do this, you will be creating dictionary files that assign bit names to characters based on the principles of Huffman Encoding. The first step to compress a text file is to determine the frequency of each character in that file. You must edit the `GenerateCode.java` file for this purpose. You can store your character frequencies in an array. Alternatively, you are free to use hashmaps and hashfunctions if you are familiar with them. Here is a reference: [Class HashMap](#). Make sure to read the comments in the `GenerateCode.java` file for specific instructions.

Once you have finished writing your code, run the `GenerateCode` file. The file takes two inputs, the location of the text file and the location of the dictionary file that will be output, respectively. Generate dictionary files for both the `franklinBio.txt` and `pennLogo.txt` files. Also generate a dictionary that is optimal for both texts. To do this, put both texts in one file and generate a new dictionary for this combined file.

Checkpoint: Show a TA your dictionary files before continuing.

Now that you have your dictionary files, run `Encoder.java` to compress your text files according to the four text-dictionary pairs outlined in the table on the next page. Refer back to the `Encoder` section and the `Running Encoder.java` portion of the `Eclipse` section (at the end of the document) if you need to.

Lastly, take the compressed version of the `franklinBio.txt` and compress it again with `franklin-Bio.dic`.

Decoding files

Once you have completed encoding the files, you can decode them using `Decoder.java`. If you are using `Eclipse`, open `Decoder.java` and set up a new run as explained in the `Eclipse` section. `Decoder.java` takes four arguments:

```
<encoded file> <code file> <output file> <number of bytes to decode>
```

For a given data file and dictionary, your encoder should report the number of bytes encoded, use that number for `<number of bytes to decode>` when decoding with the same dictionary and corresponding encoded data file.

Compression Rates

Below, you will find an explanation of each of the files that you have been using. Before you begin encoding, you should explore the `.txt` and `.dic` files in notepad.

[franklinBio.txt](#) The Autobiography of Benjamin Franklin by Benjamin Franklin from Project Gutenberg
[pennLogo.txt](#) ASCII art version of the UPenn logo

To calculate the rate of compression, take the ratio of the size of the compressed file to the size of the original file (*i.e.* $\frac{\text{compressed}}{\text{original}}$). Make sure you use the size in Bytes and not a bigger unit (*e.g.* kB, MB). A smaller ratio means better compression.

To check the encoder, you can decode the encoded file and compare it to the original. You can visually compare the files or if you want to compare byte to byte, you can use www.comparemyfiles.com.

Make sure to encode the text files according to each of the pairs of text and dictionary files listed below.

Encoding Number	Data File	Dictionary
1	franklinBio.txt	franklinBio.dic
2	pennLogo.txt	pennLogo.dic
3	franklinBio.txt	all.dic
4	franklinBio.txt	pennLogo.dic

Show your TA how you encode `pennLogo.txt` using `pennLogo.dic`. Show the compression rate achieved. Decode your encoded file and use www.comparemyfiles.com to compare the decoded file with the original file (required for **Exit Ticket**).

Lab Writeup Guidelines

Theory

Pre-lab Question:

- How many bits are in a Byte?
- Which has greater entropy and why?
 - Country of birth for U. S. presidents.

- City of birth for U. S. presidents.
- How would you convert a string of eight (8) zeros and ones into a byte value if the leftmost character is the most significant bit?
- If the end of your encoding has fewer than the number of bits needed for a byte (e.g., 1010), what must you do in order to make it a full byte? (Remember that your encoder can only write full bytes)
- What cleanup activities must you perform when you reach the end of the input file?

Analysis

Decoding files

Question 1: Why do we need to specify the number of bytes to decode for the Decoder?

Compression Rates

Question 2: What is the compression rate for each of the file-dictionary pairs from the table provided?

Question 3: Which case is compressed most? Why?

Question 4: Why does `franklinBio.txt` increase in size when encoded with dictionary `pennLogo.dic`?

Question 5: What happens to the compression rate of the dual compressed file of `franklinBio.txt`? Why?

Conclusion

Question 6: List four places where lossless compression could be employed to improve your phone / computer / communication experience. Explain the beneficial impact of each case in one or two sentences.

Further Thought

*This section is **not** part of your required assignment. Along with each week, we will offer directions and questions for further thought. Due to the nature of this course, we can only begin to glimpse the depth and richness of each of the topic areas. These questions offer some headings to contemplate further depth. These will often be open-ended questions.*

Consider how you could determine a good code (provides high compression) for a given document?
The optimal code?

Eclipse

This section will guide you through setting up Eclipse and teach you how to run your code in Eclipse. Before we begin, make sure you have a copy of [lab3.zip](#), you *do not* need to extract it.

Starting Eclipse

- From the start menu run Eclipse 3.4 (**Programs**→**Programming Languages**→**Eclipse-3.40 (Ketterer Lab)**)
- If this is the first time you run Eclipse, it will ask you to choose a directory for your workspace. The workspace is where your `.java` files will be saved. Choose an appropriate location and click OK (Fig. 1).

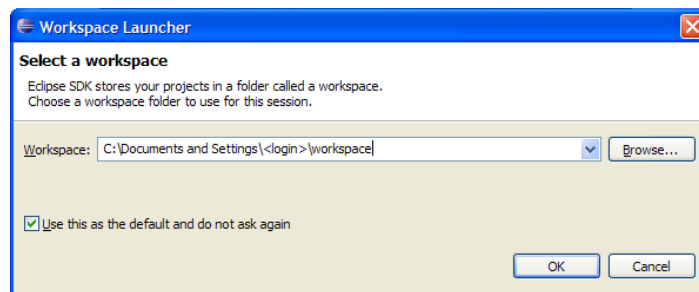


Figure 1: Workspace Launcher

- Once Eclipse starts, you can close the **Welcome** tab if it is open.

Importing a Project

Eclipse lets you work on multiple projects at a time by separating them into project directories. `lab3.zip` contains the project for this lab.

- To import a project into Eclipse, choose **File** → **Import**
- Under **General** choose **Existing Project into Workspace** (Fig. 2)
- Choose **Select archive file:** and browse for `lab3.zip` (Fig. 2)
- Under **Projects**, confirm that `lab3` is selected and choose **Finish**

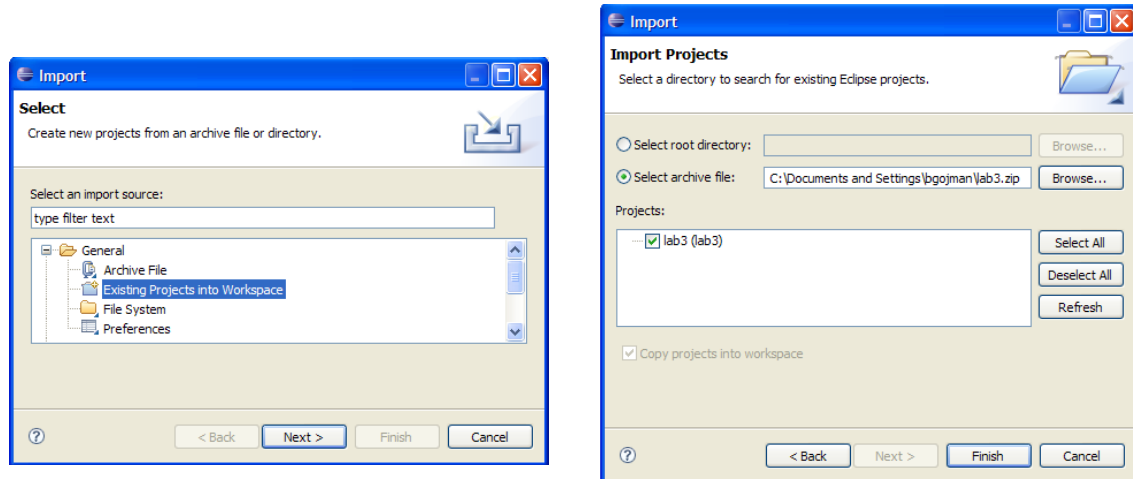
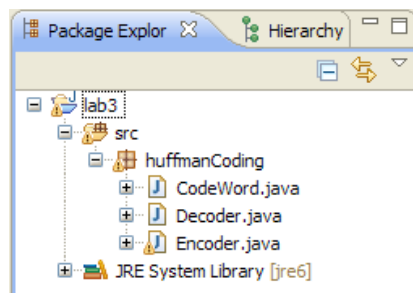


Figure 2: Import Project

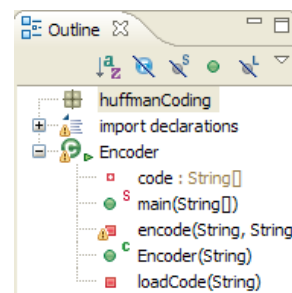
Eclipse Workspace

Once the project is loaded, you will be returned to the Eclipse workspace. Lab 3 has three files: `CodeWord.java`, `Decoder.java` and `Encoder.java`. You will primarily work with the latter.

- On the left side of Eclipse, you will find the Package Explorer with an entry for lab3. Expand lab3 until you find `Encoder.java` and double click on it (Fig. 3a).



(a) Package Explorer Tab



(b) Outline Tab

Figure 3: Eclipse Workspace

`Encoder.java` opens in the main window and on the right side the Outline tab lists its methods (Fig. 3b). You can use the Package Explorer and Outline tabs to navigate your way around your project.

You may return to the lab and return to the Running `Encoder.java` section below when you are ready to run your code.

Running Encoder.java

- To run your program the first time, make sure `Encoder.java` is the visible code file and go to **Run** → **Run As** → **Java Application**. This will run your application and also create an **Encoder** run entry in the **Run History** of the **Run** menu for future runs.

The first time you run your program, you will get a message in the console tab (under the main code window) that says:

```
Expecting 3 arguments <text file> <code file> <output file> but got 0
```

This means that `Encoder.java` expects command line arguments.

- To specify command line arguments, open the Run Configurations window (**Run** → **Run Configurations...**).
- Select *Encoder* on the left side and choose the *Argument* tab.
- In the Program Arguments text box, enter the arguments you wish to pass to your program (Fig. 4 shows the command line arguments, each between chevrons, this just indicates that they are variables you need to fill in, the final command line arguments should not have the chevrons and they should each be separated by a space).

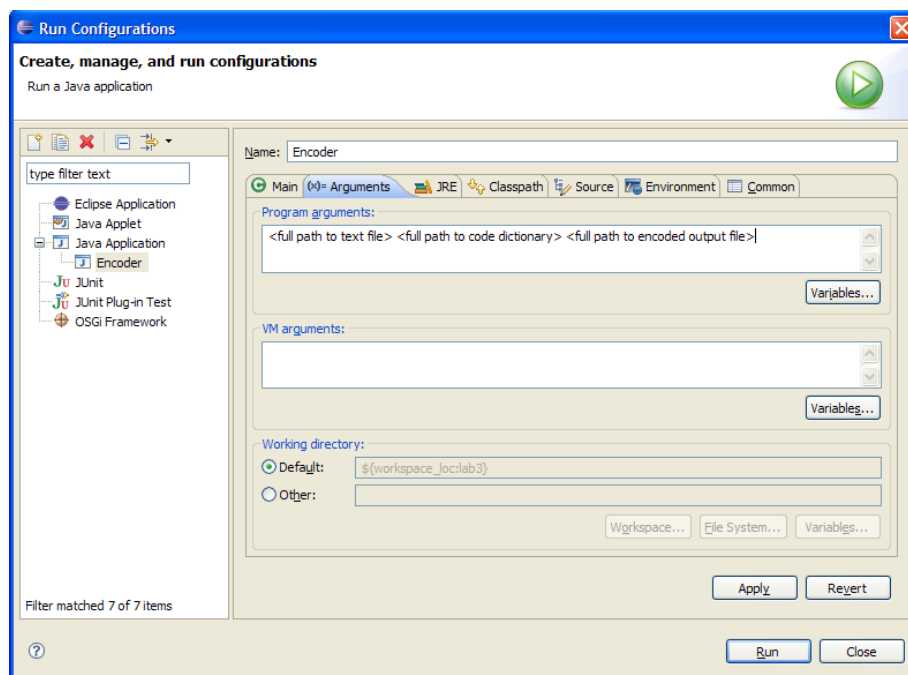


Figure 4: Adding Command Line Arguments

- Click *Run* to apply these changes and run the application or *Apply* and then *Close* to simply set the command line arguments. Either way, next time you run from the **Run History** you will run with the command line you now specified, if you want to change it, you will have to repeat these last few steps.

When you run the application, you'll see that, even with correct command line arguments, it does nothing. We will take care of that in the next section.

In this small tutorial we have barely covered the Eclipse basics. Eclipse is much more powerful than that and can help simplify code writing and organization. Feel free to explore more of Eclipse and to ask your TA if you have any questions.