Lab 3: Greedy Algorithms, Part 2.

Due date: Tuesday, April 20, at the beginning of lab period.

Lab Assignment

Assignment Preparation

This is an individual lab.

The Task

For this lab, you will conduct a comparative study of Huffman codes and ad-hoc encodings of textual data. In particular, you will do the following:

1. Implement a greedy algorithm for determining the table of Huffman codes given a textual document.
2. Implement text encoding procedure using Huffman codes.
3. Implement an ad-hoc encoding algorithm, that uses a predefined encodng table for representing characters.
4. Implement text encoding procedure using your ad-hoc encoding.
5. Develop an evaluation framework that, given a textual document, constructs both Huffman code and ad-hoc encodings of it, and evaluates the lengths of the encodings.
6. Using the developed algorithms conduct a comparative study on the benefits of using Huffman codes for encoding textual data.

Implementation of Huffman Codes

As stated above, you will implement a greedy algorithm for determining Huffman codes for a given text document. While the text documents you may receive as input may contain besides characters of Latin alphabet also various punctuation, **for this assignment you will be determining the Huffman codes only for the 26 letters of the Latin alphabet**. Your Huffman codes will be **case-insenstive**, i.e., the lowercase and the uppercase versions of the same letter will be treated as the same character. For simplicity, use `String` values to represent Huffman codes. E.g., if the Huffman code for letter "e" is 01, represent it as a string value `"01"`, not as a pair of bits. When computing sizes of encodings, you will simply treat the number of characters in the Huffman code as the number of bits.

Class CodeMap. You shall develop a Java class `CodeMap` to represent encodings of Latin alphabet characters using a code table. In your assignment, you will be using either Huffman code tables or *ad hoc* code tables. `CodeMap` instances shall represent them. The following methods shall be implemented for the `CodeMap` class.

- `CodeMap()`: this constructor creates an empty code map.
- `void assignCode(char c, String code)`: this method adds to the code map code for the character `c`. Your method shall check that `c` is in the range `a..z` (or `A..Z`, as you see fit), and complete the assignment **only** after verifying that it is so. Similarly, `code` shall be a sequence of `'0'` and `'1'` characters.
- `boolean isComplete()`: returns `true` if the `CodeMap` instance contains a code for each of the 26 characters of the Latin alphabet.
- `String convertChar(char c)`: returns the code for character `c`. If `c` is not in the code map, returns an empty string.
- `String convertText(String s)`: returns the encoding of the string `s`. Any characters **not** in the code map are skipped in the encoding.
- `void print()`: this method outputs the code map in a human readable format.

Note: You can extend the functionality of the `CodeMap` class further, as you see fit.

Class HuffmanCode. You shall create a Java class `HuffmanCode.java`. This class is responsible for generating a `CodeMap` instance representing the Huffman code table for a given input text. Instances of `HuffmanCode` class serve as `CodeMap` instance generators.

The following functionality shall be implemented:

- `HuffmanCode(String filename)`: this constructor initializes the `HuffmanCode` instance to compute Huffman codes based on the text from the file `filename`. The constructor **does not actually assign Huffman codes**, but **it does read the contents of the file** and **it does compute the character frequency table** for the input text.

- `CodeMap getHuffmanCodeMap ()`: this method implements the greedy algorithm for determining Huffman codes for a given text. It shall use the frequency statistics computed by the `HuffmanCode` class constructor described above.

Note: You can extend the functionality of the `CodeMap` class further, as you see fit.

Class HuffmanValidate. You shall implement a simple validation program `HuffmanValidate.java`. This program takes as an input parameter a name of a text file (from current directory), reads its contents, creates a Huffman code map for the text, encodes the text, and then outputs both the Huffman code map and the encoding and also prints the length of the Huffman code encoding of the text and the reduction ratio. The length shall be in bits, treating each `'0'` and `'1'` in the encoding as a single bit. The reduction ratio is between the length of the encoding in bits and the length of the original string, *also in bits*.

Use this program to establish the validity of your `HuffmanCode` implementation.

Implementation of Ad Hoc Codes

To allow for comparative analysis of the efficiency of Huffman codes, you will also implement a simple *ad hoc* encoding of a given text. The *ad hoc* encoding will uses a simple, predefined code table that will map each Latin character to a **five-bit number** representing its ordinal position in the Latin alphabet. The code table is shown below:

Ordinal	Character	Code	Ordinal	Character	Code	Ordinal	Character	Code
1	a	00001	10	j	01010	19	s	10011
2	b	00010	11	k	01011	20	t	10100
3	c	00011	12	l	01100	21	u	10101
4	d	00100	13	m	01101	22	v	10110
5	e	00101	14	n	01110	23	w	10111
6	f	00110	15	o	01111	24	x	11000
7	g	00111	16	p	10000	25	y	11001
8	h	01000	17	q	10001	26	z	11010
9	i	01001	18	r	10010			

(Note, that not all 5-bit combinations are used in this code.)

Class AdHocCode. You will create a java class `AdHocCode` which will serve as a generator class for a `CodeMap` instance representing the *ad hoc* defined above. Note, that this is done to assure the flexibility of your code. The following methods shall be implemented:

- `AdHocCode()`: this constructor will create an instance of the `AdHocCode` class.
- `CodeMap getAdHocCodeMap()`: this method shall return a `CodeMap` instance representing the ad hoc mapping above.

Class AdHocValidate. Similarly to validation of the Huffman code encoding, you will implement a simple validation program for the ad hoc encoding. The program, called `AdHocValidate.java` takes as an input parameter a name of a text file (from current directory), reads its contents, creates a ad hoc code map, encodes the text using it, and then outputs both the ad hoc code map and the encoding and also prints the length of the ad hoc encoding of the text and the reduction ratio. The length shall be in bits, treating each `'0'` and `'1'` in the encoding as a single bit. The reduction ratio is between the length of the encoding in bits and the length of the original string, *also in bits*.

Evaluation Study

Using the Huffman code and ad hoc code generation procedures, you shall conduct a study designed to illustrate the savings achieved by the use of Huffman codes vs. the use of the ad hoc encoding.

Data for the study. You will conduct the study on two datasets. The instructor will provide one dataset. You will write code creating the other dataset.

Instructor's dataset. Instructor's dataset consists of a collection of textual documents taken from Project Guttenberg. Each file represents a portion (usually the first chapter or first few chapters) of a book distributed by the Project. The list of files and the information about the texts in them is shown in the table below.

File name	Size	Book	Author	Notes
`alice.txt`	86Kb	Adventures of Alice in Wonderland	Lewis Carrol	Chapters 1–7
`grimm.txt`	180Kb	Fairy Tales	Brothers Grimm	first few tales
`gulliver.txt`	266Kb	Gulliver's Travels	Johnathan Swift	through Part 2, Chapter 7
`holems.txt`	229Kb	The Adventures of Sherlock Holmes	Arthur Conan-Doyle	first 5 stories
`pride.txt`	204Kb	Pride and Prejudice	Jane Austen	Chapters 1–20
`tomsawyer.txt`	130Kb	Adventures of Tom Sawyer	Mark Twain	Chapters 1-10
`ulysses.txt`	250Kb	Ulysses	James Joyce	first part of the novel

Your dataset. You will create a Java class `RandomText.java` which is used to generate an instance of random text. The random text generator method shall take as input an integer `Size` indicating the length of the text in bytes, and shall generate a string of this length which consists of the 26 characters of Latin alphabet.

Study 1: Huffman codes and random text.

Your first study is to check whether Huffman codes are better at encoding random text. You will write a Java program `RandomStudy.java` that combines all activities outlined below.

The goal of this study is to see whether Huffman code encodings perform better than ad hoc encodings when used to encode random strings of text.

Independent variable. The independent variable in your study is the length of the randomly generated text. This can be measured in bytes.

Dependent variables. The dependent variables in your study are the size of Huffman code/ad hoc encoding (measured in bits), and the *savings ratio*, i.e., the size of the encoding (in bits) divided by the size of the original text string (also in bits). (the smaller the ratio, the better the compression).

Experiment design. Your program shall generate random documents of the following sizes (in bytes):

128, 256, 512, 1024, 2048, 3000, 4096, 5000, 7000, 8192

For each size, your program shall generate 32 random texts of that size. For each string size, and for each string generated for it, your program shall determine the Huffman code map[1]. Using it, your program shall encode the document, and compute the length of the document and the reduction ratio. Your program then shall compute the reduction ratio for the ad hoc encoding (note, since ad hoc encoding uses constant length codes, the reduction ratio and the length of the ad hoc encoding may be computed without actually producing an encoding).

For each text size, your program shall compute and report the average values for the length of Huffman code encoding and the reduction ratio, as well as the length and the reduction ratio for the ad hoc encoding.

Report. For this study, prepare a short report that includes the table of the results to obtained, graphs your results appropriately, and provides a succinct analysis of the observed behavior. Are Huffman code encodings better or worse than ad hoc encodings for randomly generated texts?

[1]Note, you need to rerun Huffman code encoding on each document

Study 2: Huffman codes on classics

In the second study you will compare efficiency of Huffman code encodings for different classical literary works, and will rank the authors of the seven classics included in your dataset based on the reduction ratios achieved for their books.

You will write a Java program `ClassicsStudy.java`, which will facilitate this study.

Huffman Code Map preparation. For this study, you will compute Huffman code maps for each of classics separately. Each Huffman code map shall be based on the entire text fragment provided to you.

Encoding evaluation. For each text file in the input you will determine the Huffman code encoding for the entire file, **as well as for initial fragments of various sizes** of the file. In particular, for each file:

- Encode using Huffman code map **for the entire file** the first 2Kb, 4Kb and 8Kb, 16Kb, 32Kb and 50Kb of the text.
- Beyond 50Kb mark, keep adding 20Kb to the length of the encoded fragment until to reach the size of the file.

For example, for `alice.txt` you will be encoding the following initial fragments: 2Kb, 4Kb, 8Kb, 16Kb, 32Kb, 50Kb, 70Kb and the full file.

For `tomsawyer.txt` you will be encoding the following initia fragments: 2Kb, 4Kb, 8Kb, 16Kb, 32Kb, 50Kb, 70Kb, 90Kb, 110Kb and the full file.

Please note that the size of the initial fragment **shall be counted as the number of encodable characters of Latin alphabet** (that is, all digits, whitespace, punctuation, and soe on shall be ignored and not counted towards the length of the text fragment).

For each file and for each input size, you will determine the size of the Huffman code encoding of the appropriate fragment. (remember again, you will compute the Huffman codes for each file in the dataset **only once**. All fragments of a single file are encoded using the same Huffman code map.)

Report. Plot the observed behavior of Huffman code size and Huffman code reduction ratio for each author (book). (You should also plot the ad hoc encoding information as a point of reference).

Write a short report documenting this behavior. Which authors produce prose that is better encoded using Huffman codes? Would have have predicted that?

Deliverables

This part of the lab has both electronic and hardcopy deliverables. All electronic deliverables shall be submitted by the assignment deadline using the following `handin` command:

Use `handin` to submit:

```
$ handin dekhtyar-grader lab03-349 <files>
```

The hardcopy deliverables shall be submitted to the instructor during the lab period on Tuesday, April 20.

Please note, that we will start a new lab on April 13, so your work shall be completed by then (you are allowed to submit your materials, both electornic and hardcopy during that lab, but you will need to work on the next assignment during the lab.).

Electronic Deliverables. Submit all the `.java` files you produce for this assignment. Please make sure that you submit all the Java files mentioned above. Also, submit a README file containing any notes to the grader concenring the behavior of your programs and a soft copy of your report(s) in PDF format.

Hardcopy Submission. A hardcopy of your report(s) (I prefer the hardcopies submitted as a single document) should be submitted to the instructor during the lab period on April 20 (or earlier).

Note that we expect that all your programs will compile from the command line. Make sure you test for that.

Good Luck!