Huffman Coding – 100 course points

The purpose of this assignment is to practice your understanding of the tree data structure.

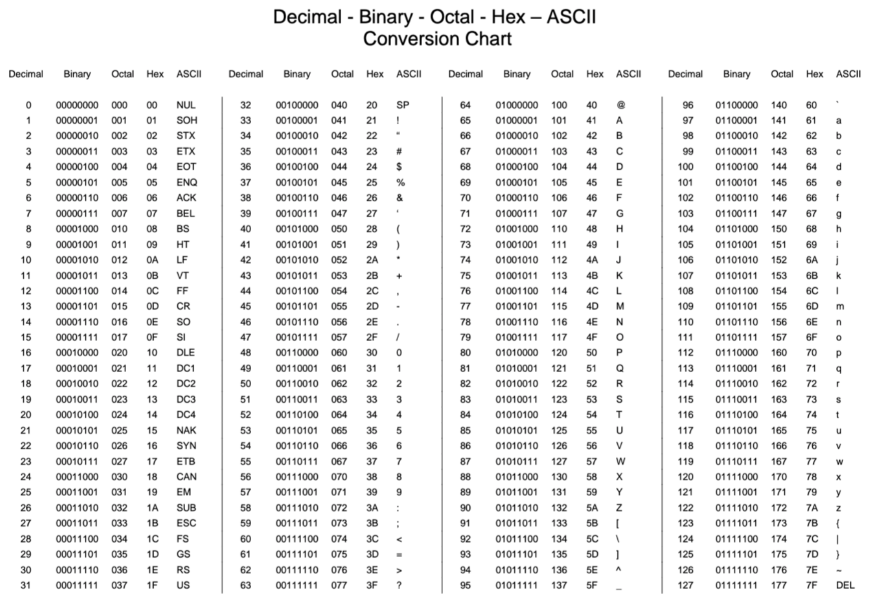
READ the assignment as soon as it comes out! You need time to understand the assignment and to answer the many questions that will arise as you read the description and the code provided.

Refer to our Programming Assignments FAQ for instructions on how to install VSCode, how to use the command line and how to submit your assignments.

Overview

The goal of this assignment is to implement a form of **data compression**. That is, given some data, we want to express the same information using less space. For this project, we will specifically focus on compressing text files, so we must first understand how computers represent text internally.

Recall that computers store data as a sequence of bytes. A byte consists of eight bits, and it represents a value between 0 and 255 inclusive. To represent English text, we need a way of assigning each English letter, punctuation symbol, special character, etc. to a sequence of eight bits (a value from 0 to 255). This mapping is provided by the **ASCII encoding**, which is shown in the table below. Notice that ASCII only uses 128 out of the 256 possible values that a byte can store.



Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

For instance, consider the text “A b ????”. Using the table above, we can see that this is represented as the following sequence of bytes: “65 32 98 32 63 63 63 63”. Note that the space counts as a character, and its value in the ASCII encoding is 32.

If we write out the binary string for each character according to the table above and concatenate them together, we get “0100000100100000011000100010000000111111001111110011111
Storing our original string with the ASCII encoding requires 8*8 = 64 bits. There are 8 characters in the text “A b ????” and each character is represented by a byte that is 8 bits long.

Now, imagine if we weren't forced to use eight bits for every character, and we could instead use the binary encoding “? = 0, [space] = 10, A = 110, b = 111”. Then our string “A b ????” would become “11010111100000”. This is only 14 bits, significantly smaller than the 64 bits that ASCII requires. Also notice that none of these codes are a prefix of any others, so there is no ambiguity when decoding. Here, we compressed our string by finding a different encoding for the characters that minimized the number of bits we needed to use. This is the essence of the **Huffman coding** algorithm, which is described in detail in the next section.

Intuitively, not only do we want to avoid wasting space encoding characters that don't appear very often, but we want to make sure that the characters which appear the most often receive the shortest codes. As you will see, **Huffman coding** is a way to do just that.

Huffman Coding

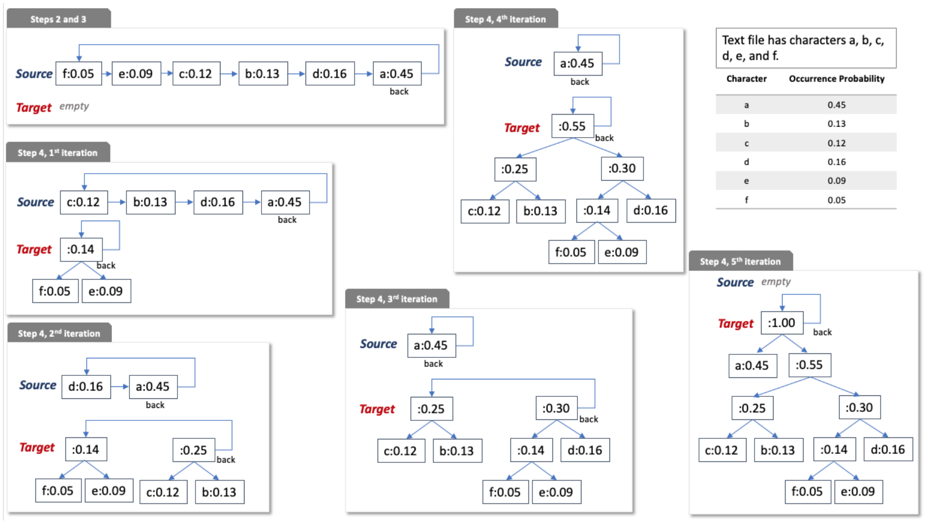
Huffman invented an algorithm that constructs the code called the Huffman code. Click here for the intuition video.

Algorithm to build the Huffman Tree. **Use this algorithm in your code.** **Click here for video explaining how to build a tree, encode and decode.**

1. Start two empty queues: *Source* and *Target*
2. Create a node for each character present in the input file, each node contains the character and its occurrence probability.
3. Enqueue the nodes in the *Source* queue in increasing order of occurrence probability.
4. Repeat until the *Source* queue is empty and the *Target* queue has only one node.
    1. Dequeue from either queue or both the two nodes with the smallest occurrence probability. If the front node of *Source* and *Target* have the same occurrence probability, dequeue from *Source* first.
    2. Create a new node whose character is *null* and occurrence probability is the sum of the occurrence probabilities of the two dequeued nodes. Add the two dequeued nodes as children: the first dequeued node as the left child and the second dequeued node as the right child.
    3. Enqueue the new node into *Target*.

Clarification notes:

- **Step 4.1** – do the following procedure twice: compare the probability occurrences of the front nodes of *Source* and *Target*. If they are equal or if *Source* is less, dequeue *Source*. If the *Target* is less, dequeue *Target*.
- **Step 4.2** – the first dequeued node and second dequeued node should be left and right children respectively.



Text file has characters a, b, c, d, e, and f.

Character	Occurrence Probability
a	0.45
b	0.13
c	0.12
d	0.16
e	0.09
f	0.05

Implementation

Overview of Files

- **CharFreq** class, which houses a **Character** object "character" representing a certain ASCII character, and a **double** "probOcc" representing its probability of occurrence (value between 0 and 1 showing its frequency). These objects are implemented to compare primarily by **probOcc**, then by **character** if those are equal. Note that "character" can be null. Getters and setters are provided. **Do not edit this class**.
- **Queue** class, which functions as a simple generic queue. It implements isEmpty(), size(), enqueue(), dequeue(), and peek(). **Do not edit this class.**
- **TreeNode** class, which houses a CharFreq object "data" representing a certain character and its frequency, and TreeNodes "left" and "right" representing the left and right subtrees of the binary tree. Getters and setters are provided. **Do not edit this class.**
- **Driver** class, which you can run to test any of your methods interactively. Feel free to edit this class, as it is provided only to help you test. It is not submitted and it is not used to grade your code.
- **StdIn** and **StdOut**, which are used by the driver, provided methods, and some of your implemented methods as well. **Do not edit these classes.**
- **HuffmanCoding** class, which contains some provided methods in addition to annotated method signatures for all the methods you are expected to fill in. You will write your solutions in this file, and it is the file which will be submitted for grading. It contains instance variables *fileName*, *sortedCharFreqList*, *huffmanRoot* and *encodings*, which must be set by your methods.
- Multiple text files which contain input data, and can be read by the driver as test cases. These files, as well as the files used for grading are guaranteed to be ASCII only. Feel free to edit them or even make new ones to help test your code. They are not submitted.

HuffmanCoding.java

NOTE: You are allowed (encouraged, even) to make helper methods in your HuffmanCoding.java file to be used in your graded methods. Just make sure that they are created with the private keyword. Do not add new imports.

Methods provided to you:

1. **writeBitString**
    - This method takes in a file name and a string consisting of the characters ‘1’ and ‘0’, and writes the string to the file.
    - You **must** use this provided method to write your encoding to your output file, and **must not** try to write your string to a file.
    - Note that it does not actually write the characters ‘1’ and ‘0’, and actually writes in bits.
    - The file name given does not need to exist yet, and if it doesn’t the method will create a new file. If the file exists, the method will overwrite it.
    - Do not edit this method.

2. **readBitString**
    - This method takes in a file name containing an encoded message, and returns a string consisting of the characters ‘1’ and ‘0’.
    - You **must** use this provided method to recover your encoded string from your input file, and **must not** try to read the encoded file yourself.
    - Note that it reads the file byte by byte and converts the bits back into characters.
    - The given file name must exist, and it must have been written to by writeBitString already.
    - Do not edit this method.

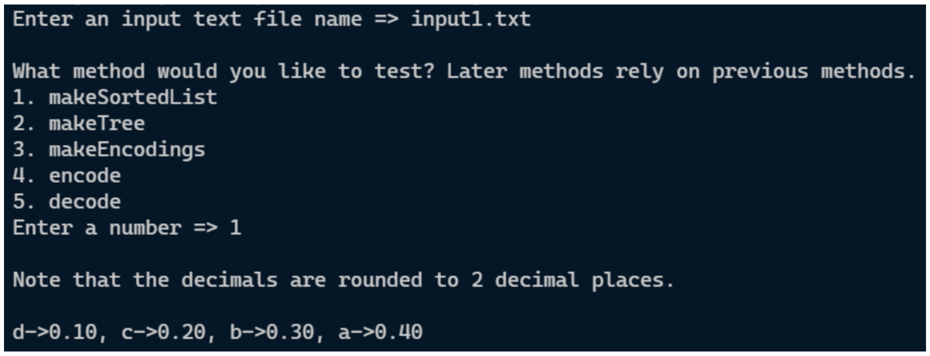
Implementation Notes

- YOU MAY only update the methods makeSortedList(), makeTree(), makeEncodings(), encode() and decode().
- **DO NOT** add any instance variables to the HuffmanCoding class.
- **DO NOT** add any public methods to the HuffmanCoding class.
- **DO NOT** use System.exit() in your code.
- **DO NOT** remove the package statement from HuffmanCoding.java
- YOU MAY add private methods to the HuffmanCoding class.

Methods to be implemented by you:

1. **makeSortedList**
    - Implement this method to read the file referenced by filename character by character, and store a **sorted ArrayList** of **CharFreq** objects, sorted by frequency, in **sortedCharFreqList**. Characters that do not appear in the input file will not appear in your ArrayList.

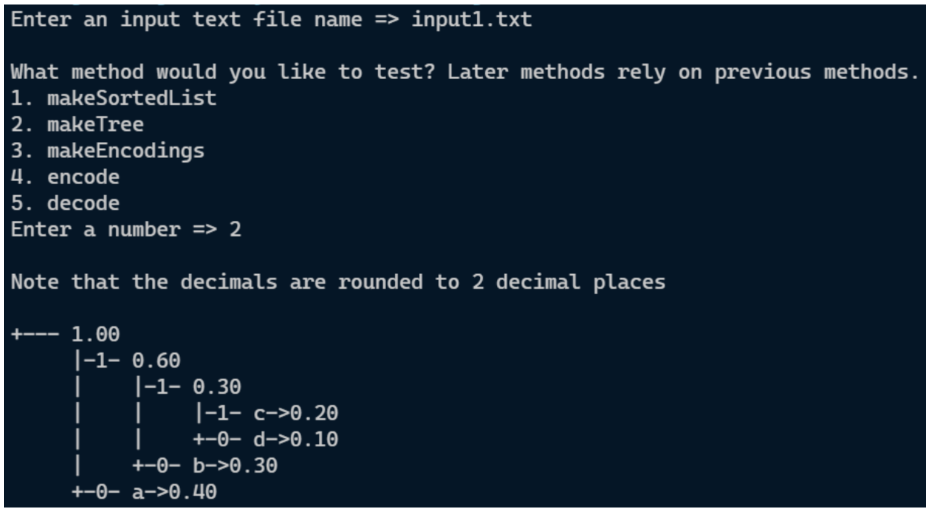
- Notice that your provided code begins by setting the file with **StdIn.** You can now use methods like **StdIn.hasNextChar()** and **StdIn.readChar()** which will operate on the file as if it was standard input.
- Also notice that there are only 128 ASCII values. This means that you can keep track of the number of occurrences of each character in an array of size 128. You can use a **char** as an array index, and it will automatically convert to the corresponding ASCII **int** value. You can convert an ASCII **int** value "num" back into its corresponding **char** with **(char) num**.
- The Huffman Coding algorithm does not work when there is only 1 distinct character. **For this specific case**, you must add a different character with **probOcc** 0 to your **ArrayList**, so you can build a valid tree and encode properly later. For this assignment, simply add the character with ASCII value one more than the distinct character. If you are already at ASCII value 127, wrap around to ASCII 0. **DO NOT** add more than one of these, and also **DO NOT** add any characters with frequency 0 in any normal input case.
- Because the **CharFreq** object has been implemented to compare based on **probOcc** primarily, you can simply use **Collections.sort(list)** before returning your final **ArrayList**. You do not need to implement your own sorting method.
- Below is an example of running the driver to help test this method.



```
Enter an input text file name => input1.txt

What method would you like to test? Later methods rely on previous methods.
1. makeSortedList
2. makeTree
3. makeEncodings
4. encode
5. decode
Enter a number => 1

Note that the decimals are rounded to 2 decimal places.

d->0.10, c->0.20, b->0.30, a->0.40
```

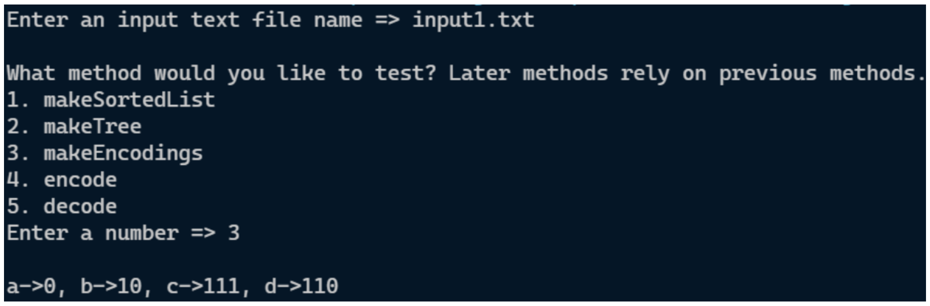
2. **makeTree**
   - Implement this method to use **sortedCharFreqList** and store the root of a valid Huffman Coding tree in **huffmanRoot**.
   - You will be using the **TreeNode** class to represent one node of your Huffman Coding tree. It contains a **CharFreq** object as its data, and references to the left and right.
   - You will be using the provided **Queue** class to code the Huffman Coding process. You must use the Huffman Coding algorithm outlined above.

- **TreeNode**s which do not have any children represent encodings for characters. These nodes of your Huffman Coding tree must contain both a “character” and a “probOcc” in their **CharFreq** object.
- TreeNodes which have at least one child do not represent encodings for characters. These nodes of your Huffman Coding tree must contain a null “character”, and their “probOcc” must be the sum of their children. The root TreeNode will have a “probOcc” of 1.0.
- Below is an example of running the driver to help test this method.



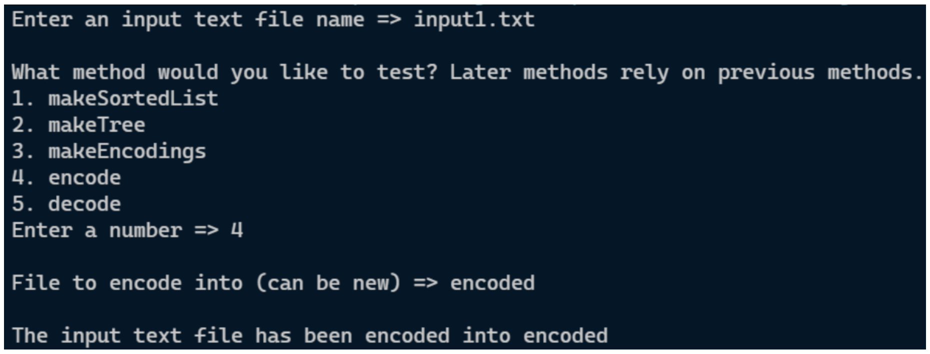
```
Enter an input text file name => input1.txt

What method would you like to test? Later methods rely on previous methods.
1. makeSortedList
2. makeTree
3. makeEncodings
4. encode
5. decode
Enter a number => 2

Note that the decimals are rounded to 2 decimal places

+--- 1.00
     |-1- 0.60
     |    |-1- 0.30
     |    |    |-1- c->0.20
     |    |    +-0- d->0.10
     |    +-0- b->0.30
     +-0- a->0.40
```

3. **makeEncodings**
   - Implement this method to use **huffmanRoot** and store a String array of size 128 containing a String of 1’s and 0’s for every character in encodings. **Character**s not present in the Huffman Coding tree will have their spots in the array left **null**.
   - Remember that going to the left child of your Huffman Coding tree represents adding a ‘0’ to your encoding for a character, and going to the right child represents adding a ‘1’. The encoding for a character is simply given by the path to get to that node from the root.
   - Remember that only **TreeNode**s with no children contain a character, and only the paths to these **TreeNode**s will receive an encoding.
   - You can convert a **Character** object to its **int** ASCII code by casting it with **(int)**.
   - Below is an example of running the driver to help test this method.



```
Enter an input text file name => input1.txt

What method would you like to test? Later methods rely on previous methods.
1. makeSortedList
2. makeTree
3. makeEncodings
4. encode
5. decode
Enter a number => 3

a->0, b->10, c->111, d->110
```

4. **encode**
    - Implement this method to use **encodings**, an input file containing some text we want to encode, and an output file we want to encode into. It will write the compressed encoding of the input file into the output file.
    - Notice that your provided code begins by setting the file with **StdIn.** You can now use methods like **StdIn.hasNextChar()** and **StdIn.readChar()** which will operate on the file as if it was standard input.
    - You are to create a **String** of ones and zeros which represents your encoding of the input text file using your encodings array. The last line of this method **must** use the **writeBitString** method to write this **String** to the file in bits. **DO NOT** try to write to the file manually.
    - If the file was successfully compressed, your output file will have a significantly smaller file size than the original text file. Feel free to open your file explorer and check!
    - Below is an example of running the driver to help test this method.

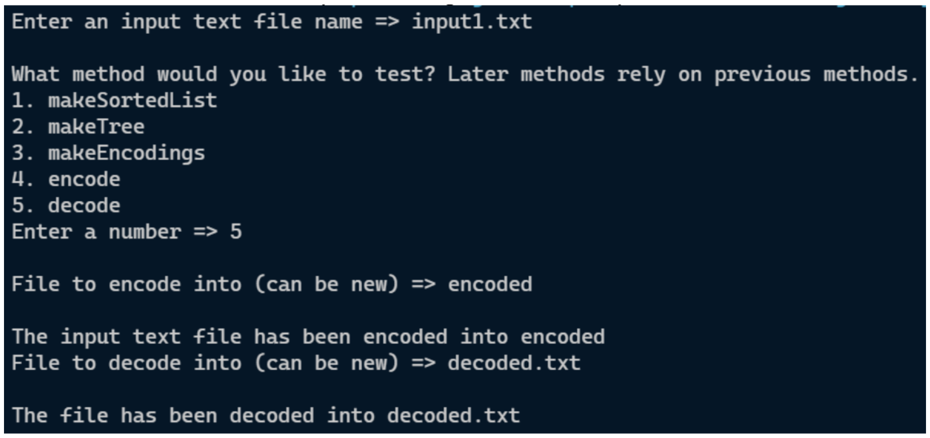


```
Enter an input text file name => input1.txt

What method would you like to test? Later methods rely on previous methods.
1. makeSortedList
2. makeTree
3. makeEncodings
4. encode
5. decode
Enter a number => 4

File to encode into (can be new) => encoded

The input text file has been encoded into encoded
```

Check out the encoded file in your file explorer. It will be significantly **smaller** than the original!

5. **decode**
    - Implement this method to take in your encoded file and use **huffmanRoot**, and print out the decoding. If

done correctly, the decoding will be the same as the contents of the text file used for encoding.
    - Notice that your provided code begins by setting the output file with StdOut. You can now use methods like **StdOut.println()** and **StdOut.print()** which will operate on the decodings file as if it was standard output.
    - You **must** start your method using the provided **readBitString** method in order to get the string of ones and zeros from the encoded file. **DO NOT** try to read the encoded file manually.
    - You must then use your tree and the procedure outlined above to decode the bit string into characters, according to the tree's encoding scheme.
    - Below is an example of running the driver to help test this method.



```
Enter an input text file name => input1.txt

What method would you like to test? Later methods rely on previous methods.
1. makeSortedList
2. makeTree
3. makeEncodings
4. encode
5. decode
Enter a number => 5

File to encode into (can be new) => encoded

The input text file has been encoded into encoded
File to decode into (can be new) => decoded.txt

The file has been decoded into decoded.txt
```

VSCode Extensions

The decoded output will be an exact copy of your input file.

You can install VSCode extension packs for Java. Take a look at this tutorial. We suggest:

- Extension Pack for Java
- Project Manager for Java
- Debugger for Java

Importing VSCode Project

1. Download the zip file from Autolab Attachments.
2. Unzip the file by double clicking it.
3. Open VSCode
    - Import the folder to a workspace through **File** > **Open Folder**

Executing and Debugging

- You can run your program through VSCode or you can use the Terminal to compile and execute. We suggest running through VSCode because it will give you the option to debug.
- How to debug your code
- If you choose the Terminal, from Huffman directory/folder:
  - to compile: **javac -d bin src/huffman/*.java**
  - to execute: **java -cp bin huffman Driver**

Before submission

Collaboration policy. Read our collaboration policy here.

Submitting the assignment. Submit *HuffmanCoding.java* separately via the web submission system called Autolab. To do this, click the *Assignments* link from the course website; click the *Submit* link for that assignment.

Getting help

If anything is unclear, don't hesitate to drop by office hours or post a question on Piazza.

- Find instructors and head TAs office hours *here*
- Find tutors office hours on Canvas -> Tutoring -> RU CATS
- In addition to office hours we have the CAVE (Collaborative Academic Versatile Environment), a community space staffed with lab assistants which are undergraduate students further along the CS major to answer questions.

Problem by Ishaan Ivaturi