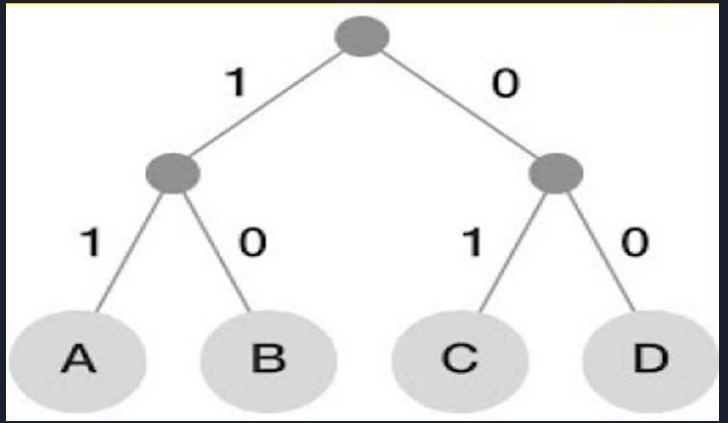
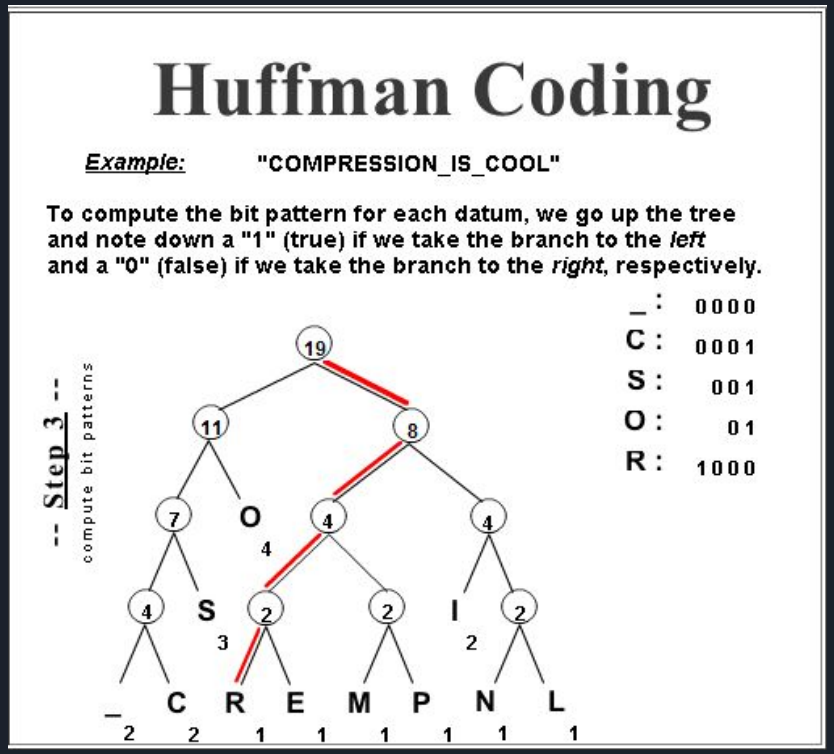
Huffman Encoding



By: Billy Dolny, Madeline DeLeon, Julia Ryan, and Savannah Rabasa

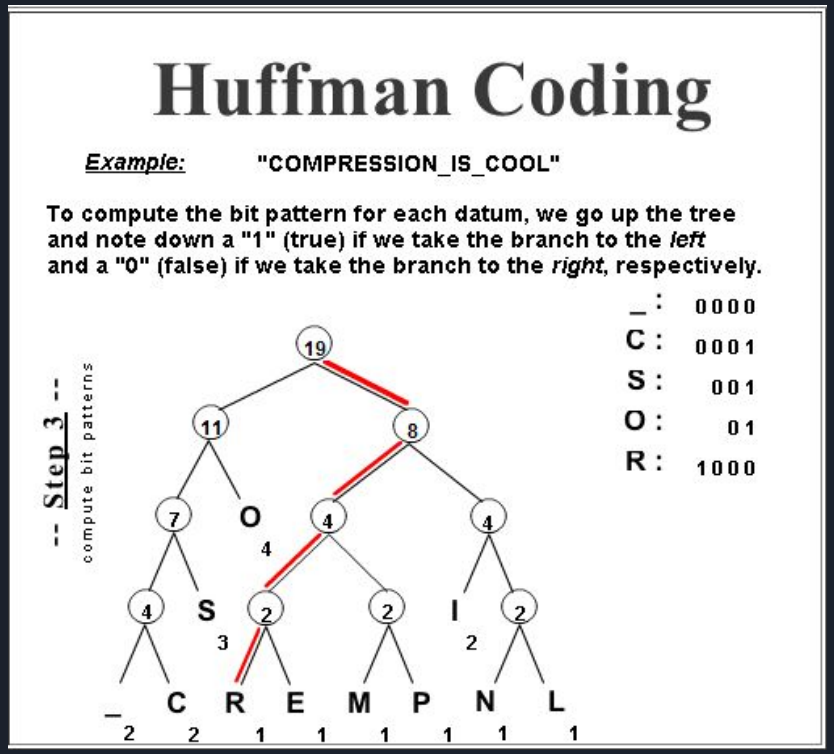
What is Huffman Encoding?

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression.



What is Huffman Encoding?

The idea is to create a search tree where each node is one element of the input file, and the nodes are ordered by how often they occur, with higher frequency nodes closer to the root. Each left branch is associated with 0, and each right branch with a 1. Then an encoding of 1s and 0s can be created to represent the original input.



Huffman Encoding in the Real World

Huffman encoding is used to compress data and make the size of the data smaller and more portable.

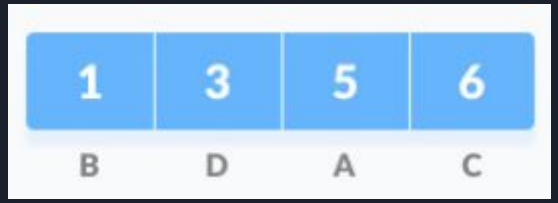
Situations that use Huffman Encoding are almost all communications to and from the internet. Most image files like jpegs and music files are Huffman encoded.

Example Walkthrough

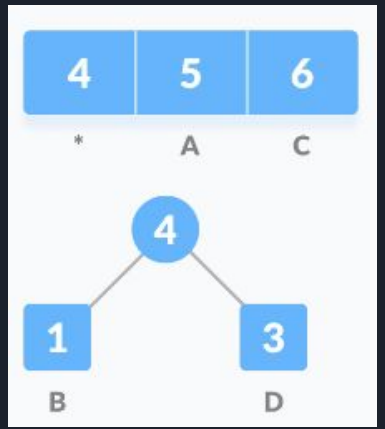
Initial String: BCAADDDCCACACAC

Each character takes up 8 bits. There are 15 characters, so the total input size is 120 bits (8 * 15)

We then calculate the frequency of each character and put that into a Priority Queue:

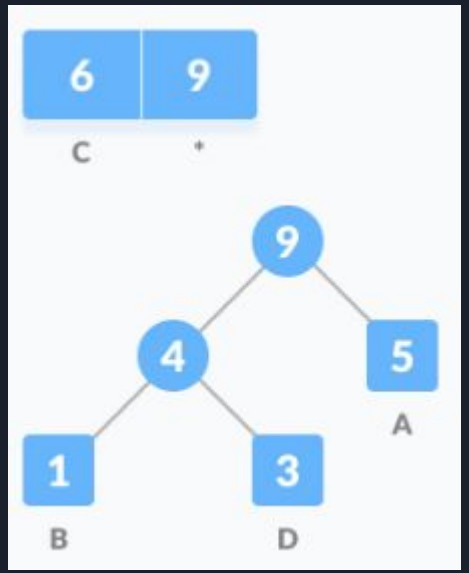


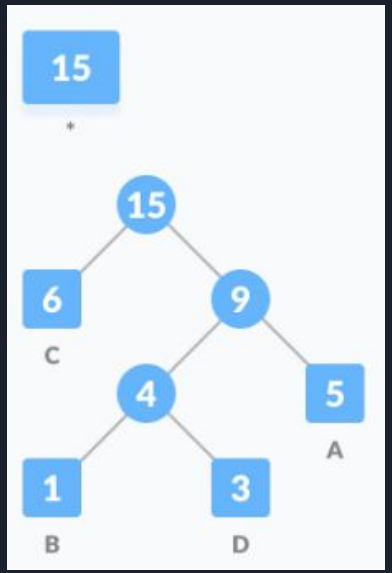
Then dequeue the two smallest nodes and make them the children of a new node that is the sum of both frequencies. Next put the new node back into the priority queue:

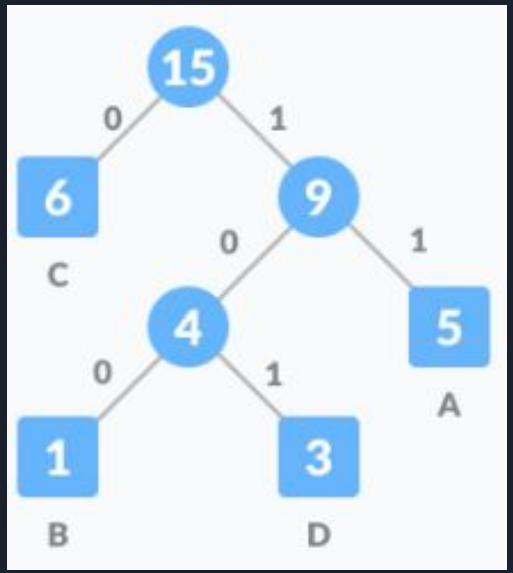


Example Walkthrough

Repeat the process over and over again until everything has been added to the tree



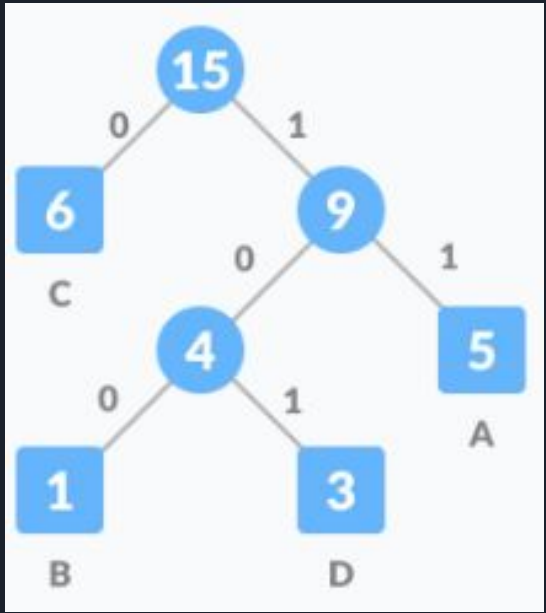




Example Walkthrough

Each leaf node is encoded in 1s and 0s. A: 11; B: 100; C: 0, D: 101

Output string: 10001111101101101001101101110



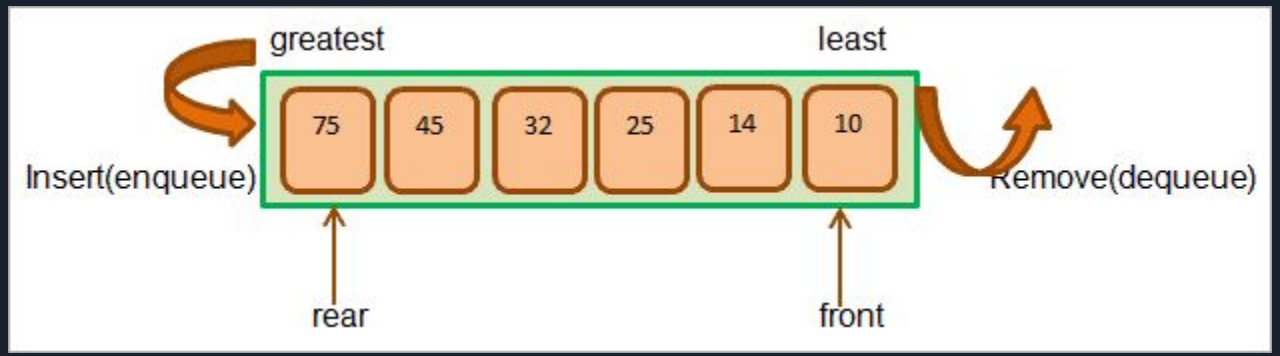
This table shows the size of the data after compression. 32+15+28 = 75 bits

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Our Implementation

We first created our own **Priority Queue.**

We coded a function that iterates through the whole input string and either enqueues a new node for a new character, or increases an existing character's frequency. The priority queue uses a min heap to ensure that the nodes with the lowest frequencies are at the front of the queue.



Creating the Huffman Tree

Next we built the **Huffman Tree** using the priority queue.

We did this by repeatedly dequeuing the two nodes with the lowest frequencies from the front of the priority queue.

Then, we created a new node whose frequency is the sum of the two current nodes. This new node becomes the parent of the two current nodes, and is inserted back into the queue.

Doing this until the queue is empty creates a tree where all internal nodes are decision points and external nodes are the actual characters that are in the input.

```
Huffman Tree: (, 15) {(C, 6) {} {}} {(, 9) {(, 4) {(B, 1) {} {}} {(D, 3) {} {}}} {(A, 5) {} {}}}
```

Encoding using Huffman Tree

Then, we created an **Encoding List** using the Huffman Tree.

We recursively traversed through the tree in order to find all of the leaf nodes. When a leaf node was found, we set the node's encoded value and added it to an Encoding list.

```
Encoding List: C: 0 // B: 100 // D: 101 // A: 11 //
```

Using this list, we can create the entire **encoded string** by iterating through each character in the input data, finding its associated binary encoding, and concatenating that onto a final string.

```
Encoded Sequence: 10001111101101101001101101110
```

Decoding from Huffman Tree

Once we have a Huffman tree and an encoded string, we can **decode** the original input data.

This is done by traversing the Huffman tree. For every 0 in the encoded string, we go to the left node, and for every 1 we go to the right node. When a leaf is hit, we add that character value from the leaf to the result string, and then start again from the root of the tree for the next number in the encoded string. This process continues until the end of the encoded string, where the resulting decoded string is then returned.

Runtime

The time complexity for setting up the priority queue and adding unique characters based on the frequency is O(NlogN). It takes N time to iterate through the entire input, and it takes logN time to make the priority queue since it uses a min heap. These events are dependent, so we multiply their runtime together.

Extracting the minimum frequency from the priority queue takes one (N/2) recursive call, which gives us the recurrence relation T(N) = T(N/2)+1; T(1)=1 and solves out to O(logN). Actually dequeuing the value is O(1), but rearranging the queue to stay in sorted order takes O(logN) (again because it uses a min heap).

The runtime for encoding the data is hard to determine because it depends on the size and shape of the Huffman tree, which depends on the size and repetition of the input data. This is also the case for decoding.

Assuming that encoding and decoding never take anymore than O(NlogN) runtime, the overall time complexity for the entire process is O(NlogN), since each step is independent.

Space

Since there isn't a standardized size and shape of the Huffman tree, the amount of space our implementation takes up is hard to determine.

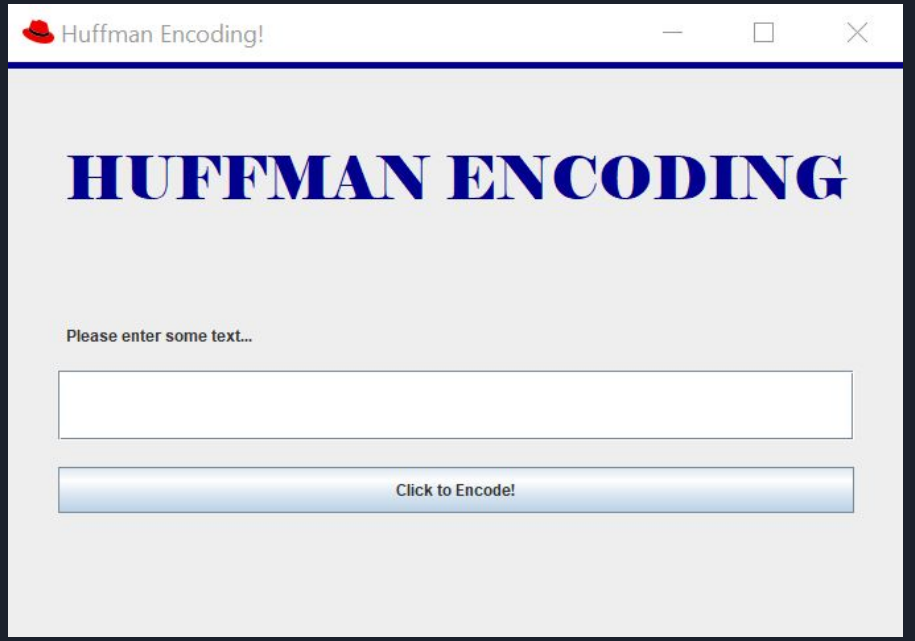
We know that the priority queue can take up no more than N space.

Generally, binary trees take up O(N) space at worst. However, for Huffman encoding, we have extra "decision point" nodes. If we assume that the tree will have no more than NlogN nodes, then the total space for our Huffman encoding is O(NlogN).

Visual GUI

The GUI has a sleek design that provides the option for the user to put in their own string and the output is sent to the console. This GUI can be used several times and will create a new output.

The GUI uses a JFrame and uses actionListeners to sent the input to the program to create the output.

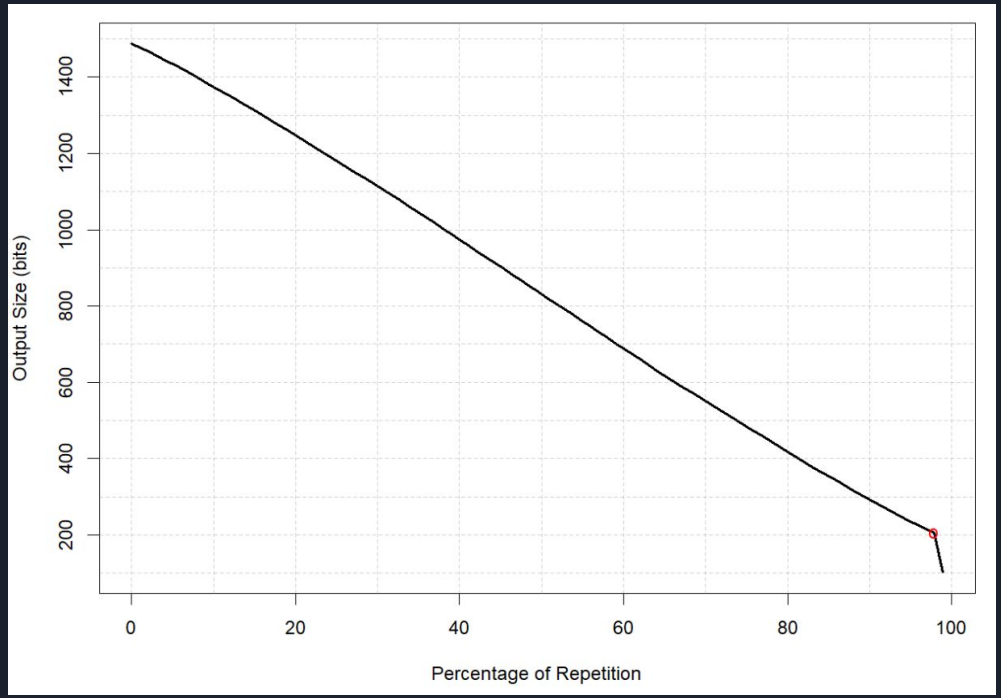


Testing and Results

Depending on the size and repetition of the input data, it may or may not be worth to use Huffman encoding to compress it. In order to determine when Huffman encoding is useful, we ran two different test sequences.

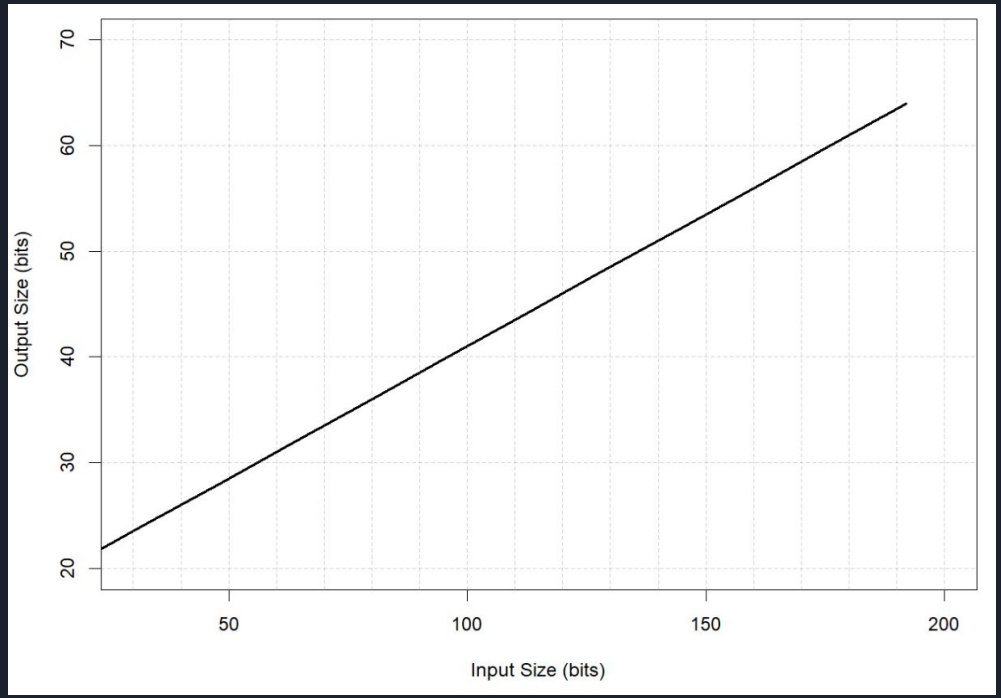
Testing and Results

The first test we ran compared the repetition of the input data to the output size of the compressed data. We made 95 testcases, which were all the same length, but had an increasing amount of repetition. As we expected, the more repetition in the input file, the smaller the output size. This graph shows the results of this experiment. You can see a significant drop in output size when every single character in the input file is the same, so the resulting huffman tree is just the root.



Testing and Results

The second test we ran compared the size of the input data to the size of the compressed output data. We made 11 testcases, which all had the same amount of repetition, but were of increasing length. As expected, the larger the input data, the larger the output size. This graph showcases those results, and clearly shows the linear relationship between the input and output size.



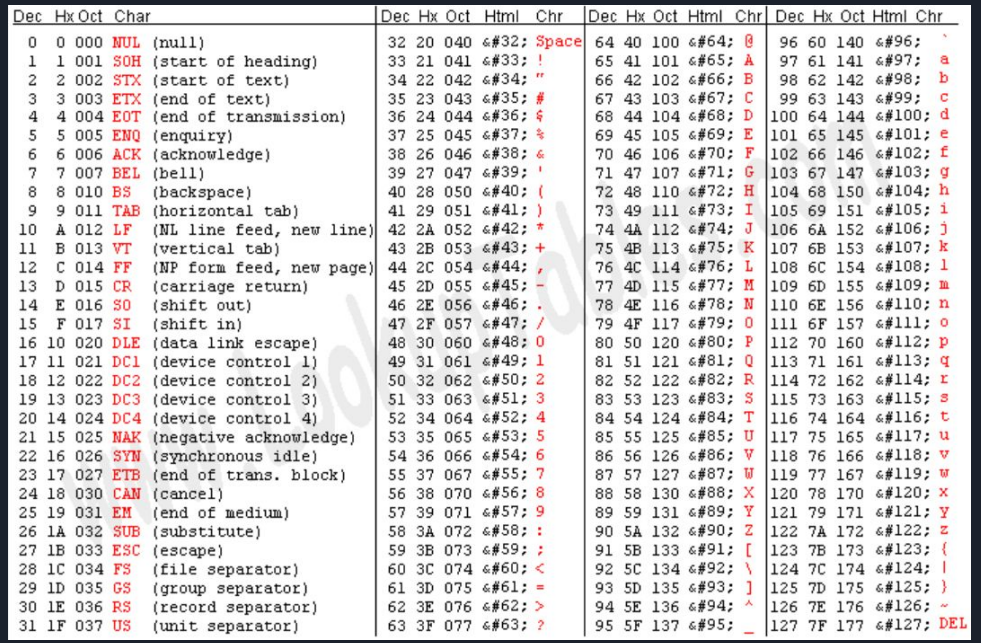
Testing and Results - Analysis

Based on the results of these two tests, we can conclude that Huffman encoding is most useful when the input data has a lot of repetition. When there's little to no repetition, the output size can be greater than the input size, which defeats the purpose of compression. Ex:

```
Input file: the quick brown fox jumps over the lazy dog
Priority Queue: {(c, 1), (x, 1), (w, 1), (j, 1), (q, 1), (z, 1), (i, 1), (h, 2), (k, 1), (b, 1), (l, 1), (y, 1), (d, 1), (n, 1), (f, 1), (
, 8), (t, 2), (m, 1), (p, 1), (s, 1), (v, 1), (r, 2), (a, 1), (o, 4), (u, 2), (e, 3), (g, 1)}
Input Size: 344
Huffman Tree: (, 43) {(, 16) {( , 8) {} {}} {(, 8) {(, 4) {(, 2) {(z, 1) {} {}} {(y, 1) {} {}}} {(h, 2) {} {}}} {(, 4) {(, 2) {(s, 1) {} {}
} {(v, 1) {} {}}} {(, 2) {(d, 1) {} {}} {(i, 1) {} {}}}}}} {(, 27) {(, 11) {(, 4) {(, 2) {(p, 1) {} {}} {(q, 1) {} {}}} {(r, 2) {} {}}} {(,
 7) {(e, 3) {} {}} {(, 4) {(, 2) {(l, 1) {} {}} {(w, 1) {} {}}} {(, 2) {(c, 1) {} {}} {(g, 1) {} {}}}}} {(, 16) {(, 8) {(o, 4) {} {}} {(,
4) {(, 2) {(k, 1) {} {}} {(m, 1) {} {}}} {(, 2) {(x, 1) {} {}} {(j, 1) {} {}}}} {(, 8) {(, 4) {(t, 2) {} {}} {(, 2) {(n, 1) {} {}} {(f, 1)
 {} {}}} {(, 4) {(u, 2) {} {}} {(, 2) {(a, 1) {} {}} {(b, 1) {} {}}}}}}
Encoding List:  : 00 // z: 01000 // y: 01001 // h: 0101 // s: 01100 // v: 01101 // d: 01110 // i: 01111 // p: 10000 // q: 10001 // r: 1001
// e: 1010 // l: 101100 // w: 101101 // c: 101110 // g: 101111 // o: 1100 // k: 110100 // m: 110101 // x: 110110 // j: 110111 // t: 11100 /
/ n: 111010 // f: 111011 // u: 11110 // a: 111110 // b: 111111 //
Encoded Sequence: 11100010110100010001111100111110111011010000111111100111001011011110100011101111001101100011011111111011010110000011000011
0001101101010010011100010110100010110011111001000010010001110110010101111
Output Size: 451
```

Testing and Results - Analysis

Since there are only 128 basic ASCII characters (256 counting non-standard characters), we know that there must be a cap for both the size of the huffman tree and how large an input file can be without containing repetition (for text-based compression). Therefore, we know that there is a certain threshold for when it's useful to use Huffman encoding, and we can assume that it's based on input size and percentage of repetition.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Challenges we faced

One challenge we had to face in the beginning was the fact we had to make a custom Priority Queue class from scratch. This took up a majority of our first week of coding since the rest of the algorithm relies on it.

We also had trouble trying to figure out how to create the GUI part of the project. There was a lot of discussion about what the GUI should do and what it should look like, as well as which Java toolkit to make it with.

Finally, in general, it took time to really understand how Huffman encoding works well enough to be able to write our own testcases and understand the results.

What we learned

During the process of creating this implementation, we learned exactly why huffman encoding is an important and useful algorithm.

We also got to see and understand why some inputs gave better results than others, and how you should only use this algorithm if you know it is going to have a lot of repetitive values (e.g. rhymes). And, we learned that if the input is relatively small, there's not really a need to compress it.

Works Cited

"Huffman Coding | Greedy Algo-3." *Geeks for Geeks*, @GeeksforGeeks, Sanchhaya Education Private Limited, www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. Accessed 15 Oct. 2023.

"Huffman Coding." *Programiz*, Parewa Labs Pvt. Ltd, www.programiz.com/dsa/huffman-coding. Accessed 15 Oct. 2023.