

NATIONAL POLYTECHNIC INSTITUTE  
SUPERIOR SCHOOL OF COMPUTER SCIENCES

ALGORITHM ANALYSIS.

## Practice 9: Huffman's Algorithm.

*Hernandez Martinez Carlos David.*  
*Burciaga Ornelas Rodrigo Andres.*

*davestring@outlook.com.*  
*andii\_burciaga@live.com.*

*Group: 2cv3.*

November 24, 2017

# Contents

<b>1</b>	<b>Introduction:</b>	<b>2</b>
<b>2</b>	<b>Basic Concepts:</b>	<b>3</b>
<b>3</b>	<b>Development:</b>	<b>4</b>
3.1	Encode: . . . . .	4
3.1.1	Symbol Frequency: . . . . .	4
3.1.2	Setting The Heap Tree: . . . . .	5
3.1.3	Assigning Codes: . . . . .	7
3.1.4	Encoding: . . . . .	8
3.1.5	Setting All Together: . . . . .	8
3.2	Decode: . . . . .	9
3.2.1	Setting The Parameters: . . . . .	9
3.2.2	Decode: . . . . .	10
3.2.3	Setting All Together: . . . . .	10
<b>4</b>	<b>Results:</b>	<b>11</b>
4.1	Test 1: . . . . .	11
4.2	Test 2: . . . . .	13
<b>5</b>	<b>Conclusion:</b>	<b>16</b>
<b>6</b>	<b>Bibliography References:</b>	<b>17</b>

# 1 Introduction:

Huffman codes compress data very effectively: savings of 20 % to 90 % are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs ( i.e., its frequency ) to build up an optimal way of representing each character as a binary string. Suppose we have a 100,000 - character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by Figure 1.0. That is, only 6 different characters appear, and the character **a** occurs 45,000 times. We have many options for how to represent such a file of information. Here, we consider the problem of designing a binary character code ( or code for short ) in which each character is represented by a unique binary string, which we call a **codeword**. If we use a **fixed-length code**, we need 3 bits to represent 6 characters:  $a = "000"$ ,  $b = "001"$  ...  $f = "101"$ . This method requires 300,000 bits to code the entire file. Can we do better?

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 1.0: A character-coding problem. A data file of 100,000 characters contains only the characters a - f, with the frequencies indicated. If we assign each character a 3 - bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long code- words. Figure 1.0 shows such a code; here the 1 - bit string 0 represents **a**, and the 4-bit string 1100 represents **f**. This code requires 224, 000 bits to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

## 2 Basic Concepts:

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem explained by *Team 2*, sometimes it does. The following steps help to develop a greedy algorithm.

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Show that if we make the greedy choice, then only one sub-problem remains.
- Prove that it is always safe to make the greedy choice. ( Steps 3 and 4 can occur in either order.)
- Develop a recursive algorithm that implements the greedy strategy.
- Convert the recursive algorithm to an iterative algorithm.

## 3 Development:

In this section we will implement the Huffman's Algorithm.

### 3.1 Encode:

To encode any file using the Huffman's algorithm we use the program provided stored in the directory **Encoder**, that has a *main* method and a class that implements this algorithm. The procedure and methods are explained below.

#### 3.1.1 Symbol Frequency:

We will denote as a *symbol*, any character that can be found in a file, the input of the program will be a file with extension *.txt*. As we can see in the method **getText** we open the file *Original.txt* found in the directory *Files* of our programs. In line 3 we get all the text that is on the file except the last line-break and in line 4, if the file it's empty, then the program will take the variable **TEST** that stores the string *"This is a test for Huffman's algorithm."*.

---

```
1 def getText ( ):
2     with ( open ( "Files/Original.txt", "r" ) ) as f:
3         txtin = f.read ( ).rstrip ( "\n" )
4         content = TEST if ( txtin == "" ) else txtin
5     return content
```

---

Once we have the input - text, we create an object of the class **Huffman** and pass to the constructor the variable returned in the method above. Then we call the method **setFrequency**, this may be the easier part of the algorithm, in line 2 we store in *total* the total length of the input string adding one more element, this will be a "flag" that will help us to know when we have reached the **END** of the file when we *decompress*. in line 4 and 5 there are declared the "dictionaries" *probability* and *frequency*, the first will store the probability of appearance of each *symbol* and the second the frequency that this *symbol* has appear. From lines 6 - 8 we fill this "dictionaries" and in lines 9 - 10 we set the **END symbol**. The dictionary that we are going to use along the program it's **frequency** the other one it's just for giving format to one of the outputs files that we will explain later.

---

```
1 def setFrequency ( self ):
2     total = len ( self.txtin ) + 1
3     c = Counter ( self.txtin )
4     self.probability = {}
5     self.frequency = {}
6     for char, counter in c.items ( ):
7         self.probability [ char ] = float ( counter ) / total
8         self.frequency [ char ] = counter
9     self.probability [ "end" ] = 1.0 / total
10    self.frequency [ "end" ] = 1
```

---

Let's imagine that we are going to encode the string **aaaabbbcc** then, we have the next appearance frequency: **a** = 4, **b** = 3 and **c** = 2. Each *symbol* has the following distribution: **P(a)** = 0.4 %, **P(b)** = 0.3 % and **P(c)** = 0.2 %, the rest it's assigned to the flag *symbol* **P(END)** = 0.1 that appears 1 time. Table 1 shows this results, it should be noted that these values are stored in the dictionaries previously mentioned.

Symbol	Distribution	Frequency
a	0.4 %	4
b	0.3 %	3
c	0.2 %	2
END	0.1 %	1

Table 1.

### 3.1.2 Setting The Heap Tree:

To encode using the Huffman's algorithm we need to set a coding *tree* with the following conditions:

1. Set all the *symbols* in a priority queue according to its frequency.
2. Combine all the *symbol* less frequents in a single node of the tree.
3. Insert the new node to the priority queue.
4. Repeat steps 2 and 3 until there's only 1 element in the priority queue.

To create a priority queue in python we import the module **heapq** that provides a priority queue based on heaps. Retaking the previously example, our priority queue shall look like in Figure 3.2.0.

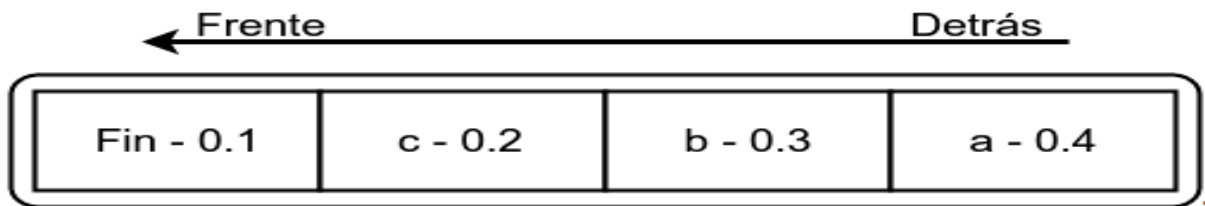


Figure 3.2.0: Priority queue for table's 1 column Frequency.

The code works as follows, from the dictionary **frequency** we extract each *symbol* with its *frequency*, then, in line 5 we append in the priority queue **self.tree** as a tuple where its first element it's the frequency, the second it's the depth level on the *heap tree* that we will create later ( notice that as it is a priority queue, the depth level initially will be of "0" ) and the last element is the respective *symbol*, this should satisfied the step 1. From lines 7 - 12 we can find a **while** loop, in lines 9 and 10 we **pop** from the priority queue the less frequents *symbols*. In line 11 we create a new node that it's a *tuple* that stores the in its first element the sum of both *symbols* frequency, in its second element stores the depth level increased by one unit, and finally a list with both *popped* tuples. Then, in line 12 we push the node in the priority queue converting it into a *tree* based on heaps. This last process corresponds to steps 2 and 3, if it's necessary, the program will repeat the steps until creating a tree based on heaps.

---

```
1 def setTree ( self ):  
2     # Add the symbols to the priority stack based on heaps.  
3     for char, freq in self.frequency.items ( ):  
4         # The stack its ordered by priority and depth.  
5         heapq.heappush ( self.tree, ( freq, 0, char ) )  
6     # Start to 'mix' contiguous symbols until the row has an element.  
7     while ( len ( self.tree ) > 1 ):  
8         # First and second less frequents symbols.  
9         e1 = heapq.heappop ( self.tree )  
10        e2 = heapq.heappop ( self.tree )  
11        node = ( e1 [ 0 ] + e2 [ 0 ], max ( e1 [ 1 ], e2 [ 1 ] ) + 1, [ e1, e2 ] )  
12        heapq.heappush ( self.tree, node )  
13    # Return the tree without the row.  
14    self.tree = self.tree [ 0 ]
```

---

The process previously explained can be visualized in Figure 3.2.1, where we take the first two less frequents *symbols*, we combine, and reinsert them in the priority queue.

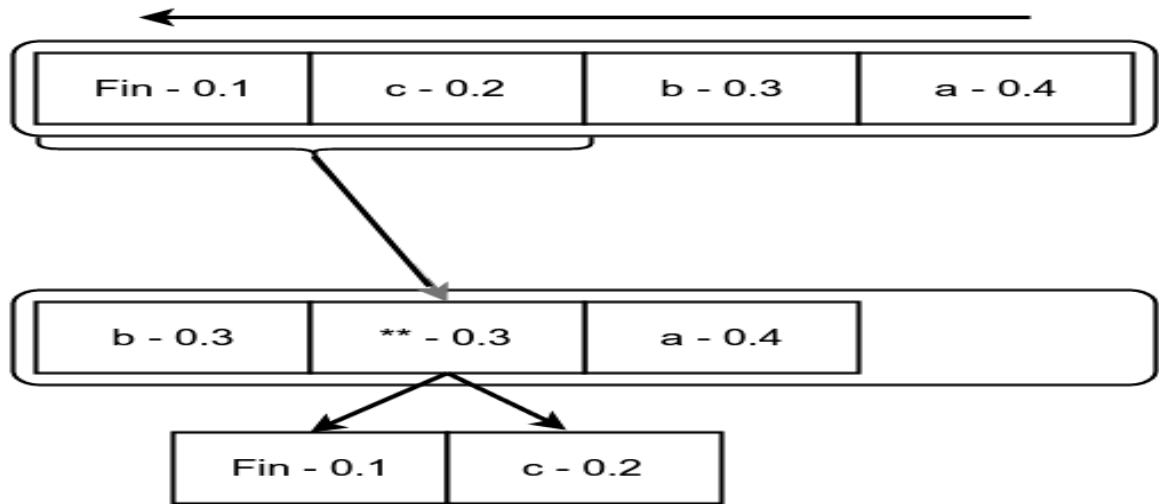


Figure 3.2.1: Steps to create the coding tree starting from the priority queue.

Finally, in line 14 we store the *heap-tree* in the same class variable that we have working with. The final result should look like in Figure 3.2.2.

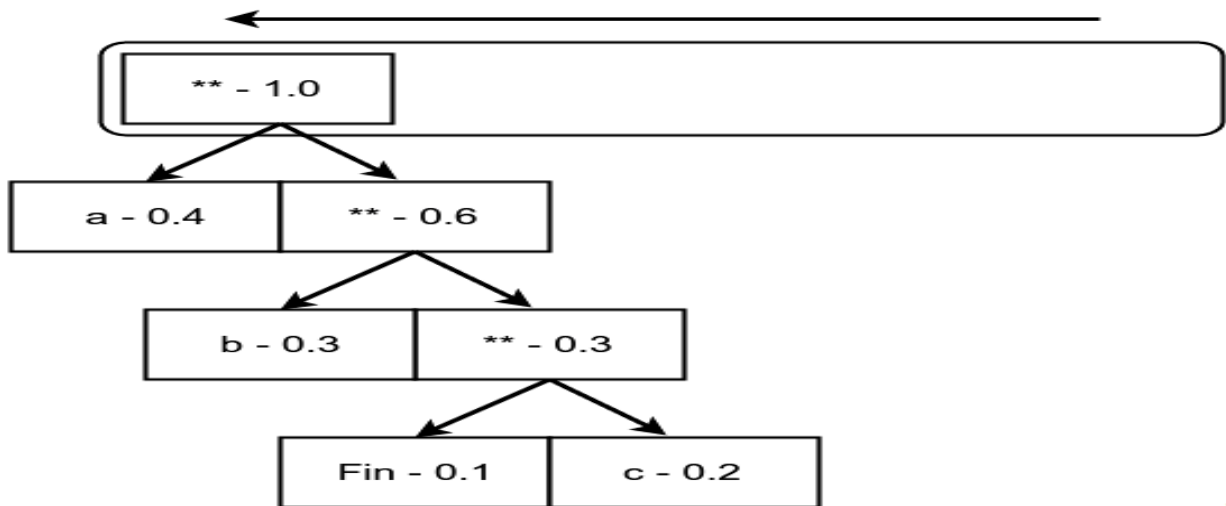


Figure 3.2.2: Result after combining all the nodes in the priority queue and generating a tree.

### 3.1.3 Assigning Codes:

When the coding tree has been created, we need to use it to create a new dictionary that will store the code of each *symbol*. To make this, we need to set a marker in each leaf of the tree, all the left and right leaves will have the labels "0" and "1" respectively, so then, each *symbol* will have as code the string of "0's" and "1's" that need to traversal to reach it as we can see in Figure 3.3.0:

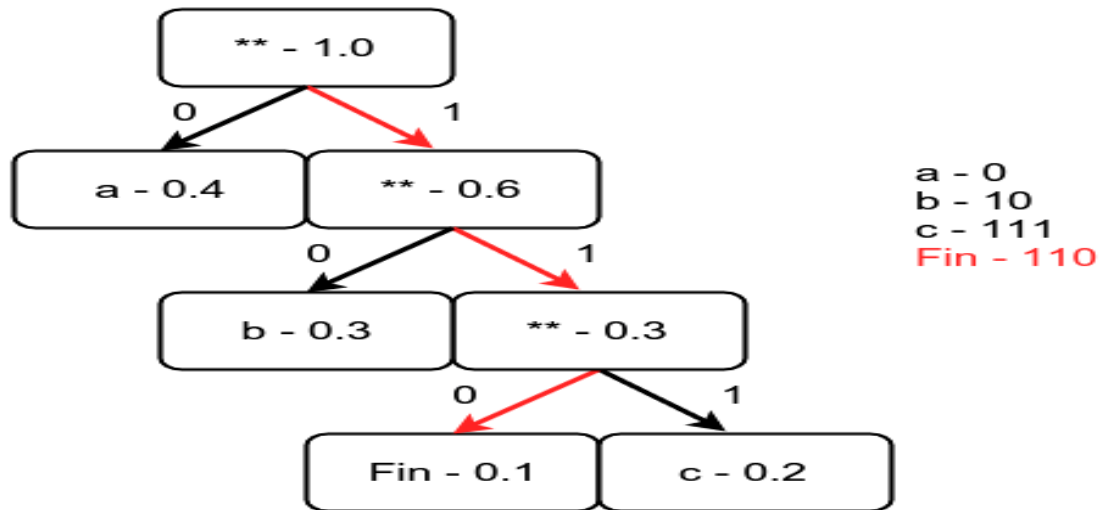


Figure 3.3.0: Binary codes for each symbol.

To make this on our code, we will use DFS ( Depth-First Search ) on the tree by generating the codes as we go down on the branches.

---

```
1 def setCodes ( self ):
2     # Depth-First Search stack.
3     searchStack = [ ]
4     searchStack.append ( self.tree + ( "", ) )
5     while ( len ( searchStack ) > 0 ):
6         element = searchStack.pop ( )
7         if ( type ( element [ 2 ] ) == list ):
8             # The node it's not a leaf.
9             searchStack.append ( element [ 2 ] [ 1 ] + ( element [ -1 ] + "1", ) )
10            searchStack.append ( element [ 2 ] [ 0 ] + ( element [ -1 ] + "0", ) )
11        else:
12            # The node it's a leaf.
13            code = element [ -1 ]
14            self.codes [ element [ 2 ] ] = code
15    pass
```

---

The code works as follows, in line 3 we declare a stack to make the DFS, in line 4 we append to the structure the tree and its "prefix" or label that we will set for this leaf, because it is the root will set the marker "". Then from lines 5 - 15 there is a **while** loop. In line 6 we will pop the last element in the stack ( FIFO ), in case that the element popped it's of *type* list it means that it's a sub-tree and has two children ( left and right nodes ), in lines 9 - 10 we will add the respectively labels to each leaf and continue with the iterations. In the other case we will ask for the respective code of that leaf in line 13 for later appended to the codes dictionary as we can see in line 14. Finally, the codes dictionary should look as:

```
self.codes = { "a": 0, "b": 10, "c": 111, "END": 110 }
```



### 3.1.4 Encoding:

Once we have the codes dictionary created, the last thing to do it's substitute each *symbol* with its respective code, so, if we have initially the input string **aaaabbbcc** then, the result should be **0-0-0-0-10-10-10-111-111-110** but, there is a problem, we don't want a string type, we want a bit string. So, we need to "cast" the result into a Integer, but if we do this, all the left "0's" will be "eliminated", so, for solving this we add to the resulting string a "1" on the first element, so then: **1-0-0-0-0-10-10-10-111-111-110**. With this we can "cast" this string into an integer without losing information. The following code make the procedure previously explained.

---

```
1 def encode ( self ):  
2     for char in self.txtin:  
3         code = self.codes [ char ]  
4         self.result = self.result + code  
5         # Add a 1 to the left and the END marker.  
6     self.result = "1" + self.result + self.codes [ "end" ]  
7     self.result = int ( self.result , 2 )
```

---

### 3.1.5 Setting All Together:

Finally we can make the respectively methods call in the order previously explained, this work it's executed by the **main** method. As we can see in the code bellow, in line 2 we get the input text, then we create an object of the class **Huffman** by passing the text as parameter in its constructor. Then in line 4 we set the frequency of appearance of each symbol. In line 5 we create the coding tree and in line 6 we extract from each leaf the respective code for its *symbol*. In line 7 we encode the input text and finally we store in our file system the resulting encoded binary string, the dictionary for decoding if it's required and the tables of frequencies and codes for each *symbol* ( This files will be presented in section 4 ).

---

```
1 def main ( ):  
2     txtin = getText ( )  
3     huffman = Huffman ( txtin )  
4     huffman.setFrequency ( )  
5     huffman.setTree ( )  
6     huffman.setCodes ( )  
7     huffman.encode ( )  
8     store ( huffman.result , huffman.frequency , huffman.probability , huffman.codes )
```

---

## 3.2 Decode:

As we have explained above, when the encoding process it's finished we store in our file system some archives, among them we can be able to find the dictionary of codes **Dictionary.dic** and the encoded text in **Encoded File.txt**. So, for decoding we need to charge this two files to the program **Decoder** already provided, and complete the following steps:

1. Read bit by bit and see if the string it's on the dictionary.
2. If the string it's on the dictionary, replace it with its respectively *symbol*.
3. Repeat the times that it's necessary.

### 3.2.1 Setting The Parameters:

As we have mention, the first step it's to charge to the program memory the *dictionary* of codes and the resulting binary string. The process work as follows, we import the module **pickle** that it's designed for binary inputs and outputs. So, in line 3 we read the *Encoded* file. In line 5 we create a variable of type FILE and read the codes dictionary, but as we are going the decode, we are going to make the inverse process, so, we create an auxiliary dictionary as we can see in line 8, this dictionary will have as *keys* the codes and as *elements* the respectively *symbols*. In lines 9 - 10 we exchange the *keys* and *elements*. Finally, the dictionary should look like this:

```
dictionary = { 0: "a", 10: "b", 111: "c", 110: "END" }
```

---

```
1 def getParameters ( ):
2     # Get the compressed binary sequence.
3     compressed = pickle.load ( open ( "../Encoder/Files/Encoded File.txt", "rb" ) )
4     # Get the dictionary with the coding of each symbol.
5     f = open ( "../Encoder/Files/Dictionary.dic", "r" )
6     dictionary = f.read ( )
7     dictionary = ast.literal_eval ( dictionary )
8     aux = { }
9     for key, element in dictionary.items ( ):
10         aux [ element ] = key
11     f.close ( )
12     return compressed, aux
```

---

### 3.2.2 Decode:

As we are going to work at bits level, we will be shifting along the input binary-string. Remember that the first element of the string it's an extra "1", so in line 3 we calculate the length of the binary-string but subtracting one unit. In line 5 we verify that the first element it's a "1" otherwise the file is corrupted. Then, we set a flag named *done* as **False** as the name of the variable, this flag will help the program to know when the decoding process is completed. In line 8 there is a **while** loop. In line 9 we create a variable *shift* that will store the number of positions that we will be *shifting* in each iteration ( we are going to analyze bit by bit ).

***Observation:** In python the binary-strings are represented like this **0b1000010101011111110** so, initially the variable *length* stores the value "19" and *shift* "18", for this example.*

In line 11 there is an infinite **while** loop, in line 12 we shift into out binary-string 18 positions to the right, so, the variable **num** should store "0b10" ( continuing with the *observation* example ), as we can see, there is a useless "0b" character and the extra "1" added in the encoding process, so in line 14 we delete this extra characters and store the result in the variable **bitnum** that at the moment should store the value of "0". Then we evaluate if **bitnum** it's in the dictionary, if isn't, then we decrease the **shift** variable and continue searching in the next iteration, otherwise we extract the respective symbol, and appended in the list **result** as we can see in line 22. Finally in lines 23 and 24 we modified the variables **binary** and **length** by preparing them for the next iteration.

***Observation:** The **while** loop will change the **done** flag until the program finds the codes for the **END** symbol.*

---

```
1 def decode ( dictionary , binary ) :
2     result = [ ]
3     length = binary.bit_length ( ) - 1
4     # First character must be 1, otherwise there is an error.
5     if ( binary >> length != 1 ) :
6         raise ( "Error: Corrupt file." )
7     done = False
8     while ( length > 0 and ( not done ) ) :
9         shift = length - 1
10        # Increase bit by bit.
11        while ( True ) :
12            num = binary >> shift
13            # Delete the initial '1' and the '0b' of the format.
14            bitnum = bin ( num ) [ 3: ]
15            if ( bitnum not in dictionary ) :
16                shift -= 1
17                continue
18            char = dictionary [ bitnum ]
19            if ( char == "end" ) :
20                done = True
21                break
22            result.append ( char )
23            binary = binary - ( ( num - 1 ) << shift )
24            length = shift
25    return "".join ( result )
```

---

### 3.2.3 Setting All Together:

Finally we can make the respectively methods call in the order previously explained, this work it's executed by the **main** method. As we can see in the code bellow, in line 2 we get the parameters for the decompression and in line 3 we make the call to the method `decode` that returns the input text decoded. Finally, in lines 4 - 5 we store the decoded text in a file with the name "Decoded File.txt".

---

```
1 def main ( ) :
2     compressed , dictionary = getParameters ( )
3     decompressed = decode ( dictionary , compressed )
4     with ( open ( "Files/Decoded File.txt" , "w" ) ) as f :
5         f.write ( decompressed )
```

---

## 4 Results:

All the code shown above doesn't have significance if its operation is not shown. This section will show you the output files generated by the Huffman's algorithm, and we will verify if it really compress the file.

### 4.1 Test 1:

The first test consist in encode an empty file using the program provided in the folder **Encoder**, as we said, if the file it's empty, the program will take the string *"This is a test for Huffman's algorithm"*. As we can see in Figure 4.1.0, the *Original.txt* file it's empty, so, we will run the program and see what's in *Frequency.txt* and *Codification.txt* files ( Figures 4.1.1 and 4.1.2 ).

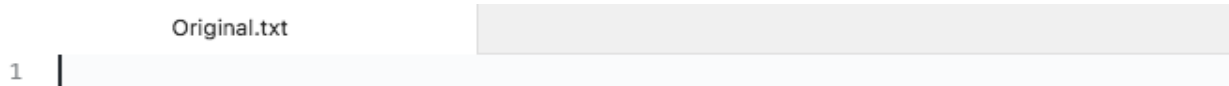


Figure 4.1.0: Empty Original.txt file.

Frequency.txt				Codification.txt		
1				1		
2	Symbol	Probability	Frequency	2	Symbol	Codes
3	T	0.025	1	3	h	0000
4	h	0.05	2	4	m	0001
5	i	0.075	3	5	o	0010
6	s	0.1	4	6	r	0011
7		0.15	6	7	'	01000
8	a	0.075	3	8	.	01001
9	t	0.075	3	9	H	01010
10	e	0.025	1	10	T	01011
11	f	0.075	3	11	e	01100
12	o	0.05	2	12	end	01101
13	r	0.05	2	13	g	01110
14	H	0.025	1	14	l	01111
15	u	0.025	1	15	n	10000
16	m	0.05	2	16	u	10001
17	n	0.025	1	17	a	1001
18	'	0.025	1	18		101
19	l	0.025	1	19	f	1100
20	g	0.025	1	20	i	1101
21	.	0.025	1	21	t	1110
22	end	0.025	1	22	s	1111
23				23		

Figure 4.1.1: Frequency.txt file for the string *"This is a test for Huffman's algorithm"*.

Figure 4.1.2: Codification of each *symbol* for the string *"This is a test for Huffman's algorithm"*.

The output text or *binary-string* generated by the encoder program it's presented in Figure 4.1.3, as we have previously mentioned, we import the python module **pickle** that server for binary inputs and outputs.

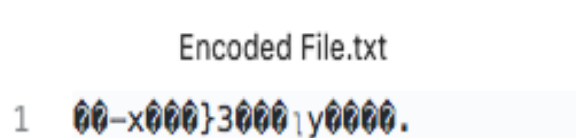
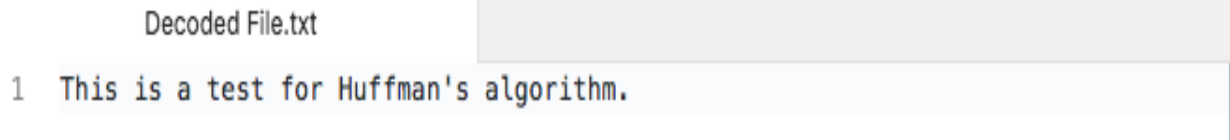


Figure 4.1.3: Encoded File.txt

For decoding we run the program provided in the folder **Decoder** that will take as parameters what ever it is in the file presented in Figure 4.1.3 and the dictionary of codes. Finally, after executing the decoding process, the program will generate a file named *Decoded File.txt*, as you can imagine, will store the decoded text. Figure 4.1.4 shows this output.



```
Decoded File.txt
1 This is a test for Huffman's algorithm.
```

Figure 4.1.4: Decoded File.txt

Finally, to see if the encoder program actually compress the size of the string *"This is a test for Huffman's algorithm"* with the help of the *system information* we visualize that the size of the decoded file it's bigger ( Figure 4.1.5 ) than the encoded one ( Figure 4.1.6 ).

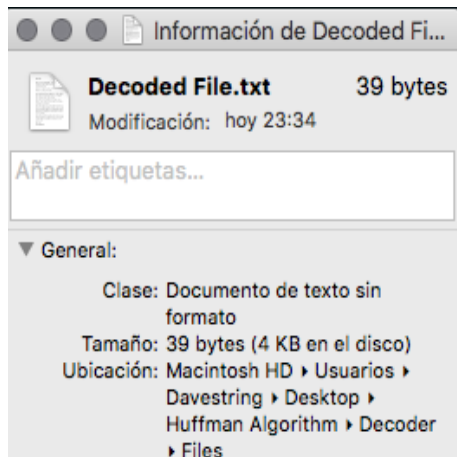


Figure 4.1.5: Decoded file size.

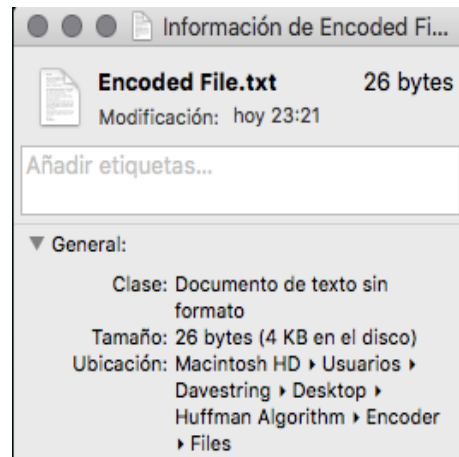


Figure 4.1.6: Encoded file size.

As we can see the Decoded file has a size of 39 bytes in comparison with the Encoded one that has a size of 26 bytes.

## 4.2 Test 2:

Now let's encode a bigger file. As we can see in Figure 4.2.0, the *Original.txt* file has a lot of text, so, we will run the program and see what's in *Frequency.txt* and *Codification.txt* files ( Figures 4.2.1 and 4.2.2 ).

```

Original.txt
1  #include "Library.h"
2
3  void *ProducerZero ( void *argument ) {
4
5      struct sembuf buf;
6      char string_a [ 20 ];
7      int *ThreadNumber;
8      int counter = 1;
9
10     ThreadNumber = ( int * ) argument;
11     strcpy ( string_a, "aaaaaaaaaaaaaaaaaaaaa" );
12
13     do {
14         for ( int secondCounter = 0 ; secondCounter < 10 ; secondCounter++ ) {
15             Wait ( &buf, secondCounter );
16             if ( blocks [ secondCounter ].identifier == -1 ) {
17                 Wait ( &buf, 11 ); // Productor
18                 blocks [ secondCounter ].identifier = *ThreadNumber;
19                 strcpy ( blocks [ secondCounter ].string, string_a );
20                 printf ( "\na = %d.", counter );
21                 counter++;
22                 Signal ( &buf, 10 );
23             } /* End of if. */
24             Signal ( &buf, secondCounter );
25             if ( counter > Limit ) {
26                 pthread_exit ( NULL );
27             } /* End of if. */
28         } /* End of for. */
29     } while ( counter <= Limit );
30
31     //printf ( "\n\n\tThread Number: %d.\tString: %s.", *ThreadNumber, string_a );
32
33     return NULL;
34
35 } /* End of ProducerZero Threads. */
36

```

Figure 4.2.0: Original.txt that store a C program.

Frequency.txt							
	Symbol	Probability	Frequency				
1				32	{	0.0049	5
2				33	s	0.0219	22
3	#	0.0009	1	34	f	0.0199	20
4	i	0.0359	36	35	;	0.0199	20
5	n	0.0479	48	36	_	0.0049	5
6	c	0.0239	24	37	[	0.0039	4
7	l	0.0069	7	38	2	0.0009	1
8	u	0.0319	32	39	0	0.0039	4
9	d	0.0299	30	40	]	0.0039	4
10	e	0.0479	48	41	T	0.0059	6
11		0.2497	250	42	N	0.0069	7
12	"	0.0079	8	43	=	0.0079	8
13	L	0.0069	7	44	1	0.0059	6
14	b	0.0149	15	45	p	0.0049	5
15	r	0.0549	55	46	,	0.0089	9
16	a	0.0389	39	47	C	0.0079	8
17	y	0.0029	3	48	<	0.0019	2
18	.	0.0109	11	49	+	0.0039	4
19	h	0.0099	10	50	w	0.0019	2
20				51	&	0.0039	4
21		0.0339	34	52	k	0.0029	3
22	v	0.0019	2	53	-	0.0009	1
23	o	0.0389	39	54	/	0.0119	12
24	*	0.0139	14	55	\	0.0049	5
25	P	0.0029	3	56	%	0.0029	3
26	Z	0.0019	2	57	S	0.0029	3
27	(	0.0149	15	58	}	0.0049	5
28	g	0.0099	10	59	E	0.0039	4
29	m	0.0099	10	60	>	0.0009	1
30	t	0.0429	43	61	x	0.0009	1
31	)	0.0149	15	62	U	0.0019	2
32	,			63	w	0.0009	1
				64	:	0.0019	2
				65	end	0.0009	1

Figure 4.2.1: Frequency.txt file for Figure 4.2.0.

Codification.txt					
1			32	0	01110110
2	Symbol	Codes	33	E	01110111
3	}	0000000	34	[	01111000
4	!	0000001	35	]	01111001
5	/	000001	36	<	011110100
6	c	00001	37	U	011110101
7	e	0001	38	W	011110110
8	n	0010	39	Z	011110111
9	T	0011000	40	v	011111000
10	%	00110010	41	#	0111110010
11	P	00110011	42	-	0111110011
12	S	00110100	43	2	0111110100
13	k	00110101	44	>	0111110101
14	y	00110110	45	end	0111110110
15	x	001101110	46	w	0111110111
16	:	001101111	47	,	0111111
17	*	001110	48		10
18	L	0011110	49		
19	N	0011111	50		11000
20	r	0100	51	i	11001
21	d	01010	52	a	11010
22	(	010110	53	o	11011
23	)	010111	54	;	111000
24	b	011000	55	f	111001
25	l	0110010	56	g	1110100
26	"	0110011	57	h	1110101
27	u	01101	58	m	1110110
28	=	0111000	59	\	11101110
29	C	0111001	60	_	11101111
30	&	01110100	61	t	11110
31	+	01110101	62	p	11111000
--	-	-----	63	{	11111001
			64	.	1111101
			65	s	111111

Figure 4.2.2: Codification.txt file for Figure 4.2.0.

The output text or *binary-string* generated by the encoder program it's presented in Figure 4.2.3.

```

Encoded File.txt
1 000000_00:0Mq0"40i0|00N'010000(0|0b0_0w0d00?
2 00>J0100Y000]2y00W200000000Au0wwq0i0{U01~00={0f0K0A:00}0}0V3N @00<0o000T000000>
3 00w000_U0{010000K0A0<0*0/)000S0C00000i)0B04000~0wJ0t000K0000atZ'0300*000(00TU1~
4 I0000o0W2000>0200 _U0000.0000.000Y0K09%;0/0M02LK0@0WU08fo000G090^B0~0^00)0!~00k
5 a0000Mm010/y0000Y0SU00^000"00K000?0K09%;0/0M02L0<0*0/)000S0C00000i0080` :
6 /m00000v0I00000N0^00)0!~0d-i0080c0r000Zk000Zk000000t00-m00_10%00N0KG/YK000}00cc
7 /m0d08fo000G/Y100}0000000000x000000F0W00000F'000200G004000C000!MX000dA0|.

```

Figure 4.2.3: Encoded File.txt

For decoding we run the program provided in the folder **Decoder** that will take as parameters what ever it is in the file presented in Figure 4.2.3 and the dictionary of codes. Finally, after executing the decoding process, the program will generate a file named *Decoded File.txt*, as you can imagine, will store the decoded text. Figure 4.2.4 shows this output.

```

Decoded File.txt
1  #include "Library.h"
2
3  void *ProducerZero ( void *argument ) {
4
5      struct sembuf buf;
6      char string_a [ 20 ];
7      int *ThreadNumber;
8      int counter = 1;
9
10     ThreadNumber = ( int * ) argument;
11     strcpy ( string_a, "aaaaaaaaaaaaaaaaaaaaa" );
12
13     do {
14         for ( int secondCounter = 0 ; secondCounter < 10 ; secondCounter++ ) {
15             Wait ( &buf, secondCounter );
16             if ( blocks [ secondCounter ].identifier == -1 ) {
17                 Wait ( &buf, 11 ); // Productor
18                 blocks [ secondCounter ].identifier = *ThreadNumber;
19                 strcpy ( blocks [ secondCounter ].string, string_a );
20                 printf ( "\na = %d.", counter );
21                 counter++;
22                 Signal ( &buf, 10 );
23             } /* End of if. */
24             Signal ( &buf, secondCounter );
25             if ( counter > Limit ) {
26                 pthread_exit ( NULL );
27             } /* End of if. */
28         } /* End of for. */
29     } while ( counter <= Limit );
30
31     //printf ( "\n\n\tThread Number: %d.\tString: %s.", *ThreadNumber, string_a );
32
33     return NULL;
34
35 } /* End of ProducerZero Threads. */

```

Figure 4.2.4: Decoded File.txt

Finally, to see if the encoder program actually really compress the size of *Original.txt* with the help of the *system information* we visualize that the size of this file it's bigger ( Figure 4.2.5 ) that the encoded one ( Figure 4.2.6 ).

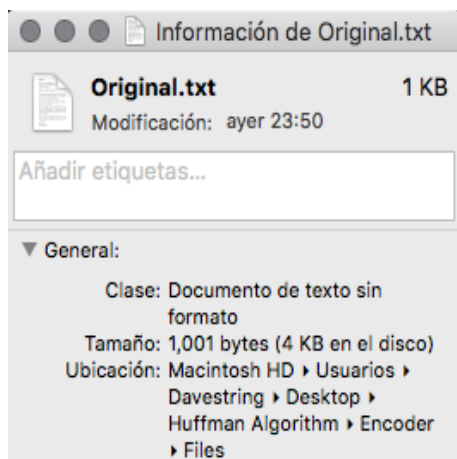


Figure 4.2.5: Decoded file size.

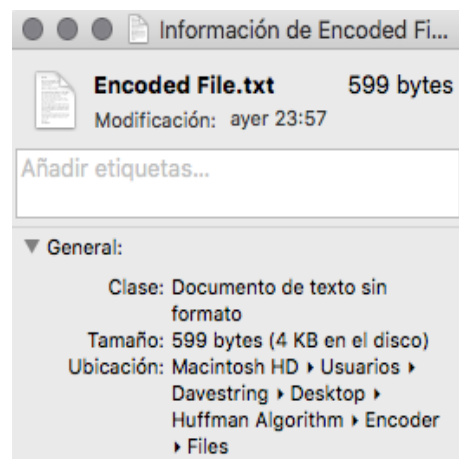


Figure 4.2.6: Encoded file size.

As we can see the *Original.txt* file has a size of 1001 bytes in comparison with the Encoded one that has a size of 599 bytes.



## 5 Conclusion:

It's a very interesting thing to see a Greedy algorithm in action. This time we had the opportunity to program the **Huffman's algorithm** that it's very useful to compress bigger files into a smaller ones saving between a 20 % and 90 % efficiently. This allow us to had an idea how compression programs like *winrar* works.

## 6 Bibliography References:

- [ 1 ] Baase and Van Gelder. "Computer Algorithms: Introduction to Design and Analysis". Addison-Wesley.
- [ 2 ] Thomas H. Cormen. "Introduction to Algorithms". The MIT press.