

**DSA Project:**

***Huffman Encoding & Decoding***

**Submitted To:**

***Dr. Naveed Akhtar***

**Submitted By:**

***Subhan Amjad***

***Muhammad Umer Riaz***

***Nabeel Ahmed***

## Introduction:

This semester we were assigned with the task to search for a project for our course '**Data Structures & Algorithm**'. The project that we chose can handle content by compressing and decompressing it. The method behind this execution lies in Huffman Coding, a strategy that employs Huffman trees to create arbitrary codes for each unique character. This program takes as text file, encodes it to make a file that takes less space than the original one. Then the program takes the encoded file as input, decodes it and stores it in a file and this decoded file is the same as the original one.

## Data Structures Used:

Our project mainly uses two data structures:

1. Binary Trees (Huffman Trees)
2. Priority Queues

## What We Learned:

Our journey started with the concept of information compression and how it is performed. We at that point inspected Huffman trees which was a possible solution. It can be seen that this tree plays an important part in creating the parallel codes of characters that regularly show up within the file. We are going to learn how this tree translates compressed content into decompressed one.

## How to use:

The usage is quite basic. Copy path files for the original, encoded and decoded files and write **.txt** after the file directory in the arguments of the constructor of the object of **huffman** class.

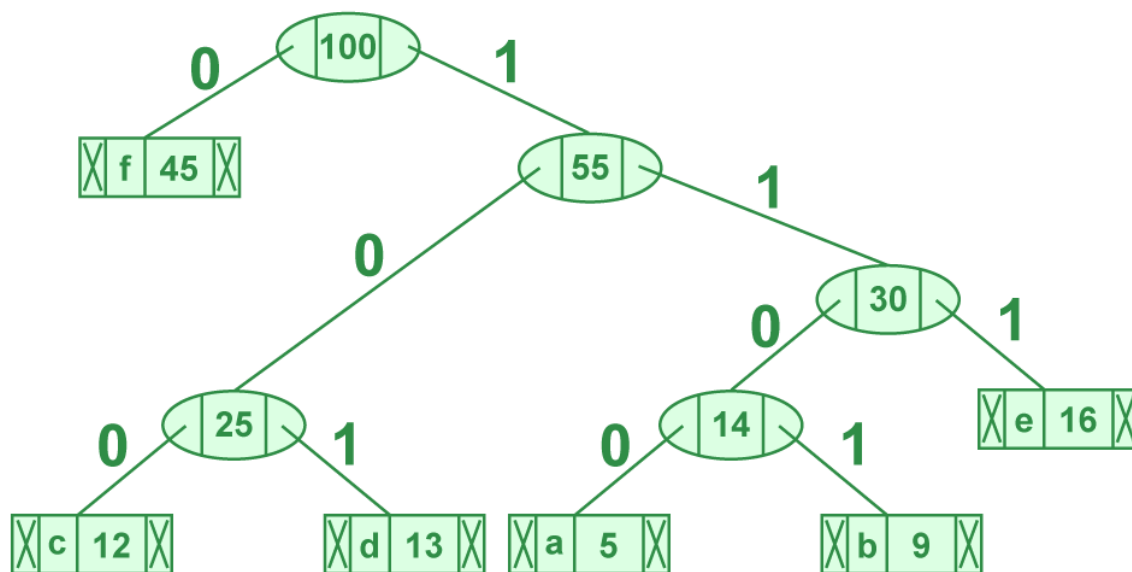
## Notes:

1. Make sure that the original file that is to be encoded does not contain any special characters other than the usual 128-ASCII characters because they are not supported by our program.
2. Only use alphabets and digits because for some characters other than the mentioned ones, it might not work.
3. Use '\\ ' instead of '\ ' in the file directory to avoid any errors instead.

## Huffman Coding:

Huffman coding is a lossless data compression algorithm that has a greedy approach. It is used in text compression, image compression and many other applications. Using binary trees, we assign codes to each unique character and use the codes instead of the original characters which is significantly smaller in size. As we move down the tree, we assign **0** to the left subtree and **1** to the right subtree. In this way, each character has its own unique Huffman code.

## Huffman Tree:



## Time Complexity:

The time complexity of Huffman Coding is  $O(n \log n)$ , where  $n$  is the number of unique characters in the original file.

## Project Roadmap:

Our journey began on 1<sup>st</sup> December, 2023.

**Project Selection:** This phase was about 2 days. During this phase we looked at many projects and all team members presented their own idea. After knowing about Huffman Coding, we were quite curious to learn and implement this data compression algorithm so we chose this project with the approval of each group member.

**Understanding:** In this phase, we understood about the working of Huffman Coding through binary trees and priority queues so that it would be much easier to implement this project if we knew a thing or two before starting the implementation. We also assigned each member their respective role in this project. This phase was completed in about 3 days.

**Algorithm:** After understanding, we moved onto the algorithm design part. This phase was a little tough as the project was completely different than the ones that we have completed before so its algorithm designing took us about 8 days. We also made flow charts and class diagrams so that it would be much easier to code

**Coding:** After algo design, we moved onto the coding part and for coding we used CodeBlocks IDE. Coding took us about 5 days.

**Testing:** We tested our code with different file sizes so ensure that our program was working fluently. We used compression ratio to find out the effectiveness of our encoding algorithm. Testing phase was completed in two days.

## **Roles of Team Members**

**Subhan Amjad** – Algorithm Design and debugging

**Muhammad Umer Riaz** – Coding, adding comments and maintaining syntax of code

**Nabeel Ahmed** – Coding and documentation

## **Algorithm**

### **Huffman Encoding:**

**Step 1:** Initialize an array of nodes for each ASCII character.

For each  $i$  from 0 to 127:

    Create a new node with  $id = i$  and  $frequency = 0$ .

**Step 2:** Take input as a text file.

**Step 3:** Read the file character by character.

For each character  $c$ :

    Increment the frequency of  $node[c]$ .

**Step 4:** Create a min heap.

**Step 5:** Insert nodes with non-zero frequency into the min heap.

**Step 6:** Create the Huffman tree from the min heap.

While the heap is not empty:

- Create a new node as the root.
- Take the top two nodes from the heap (with minimum frequency) and make them the left and right children of the root, such as left child will be the minimum frequency node and vice versa.
- Pop these nodes from the heap
- Set the frequency of the root node as the sum of the frequencies of its children.
- Push the root node back into the min heap.

**Step 7:** Assign Huffman codes to each character.

For each character c:

- Traverse the tree from the root to the leaf node corresponding to c.
- Assign '1' for each right edge and '0' for each left edge.
- Set the code of the leaf node as the assigned code.

**Step 8:** Save the binary codes to a file.

## Huffman Decoding:

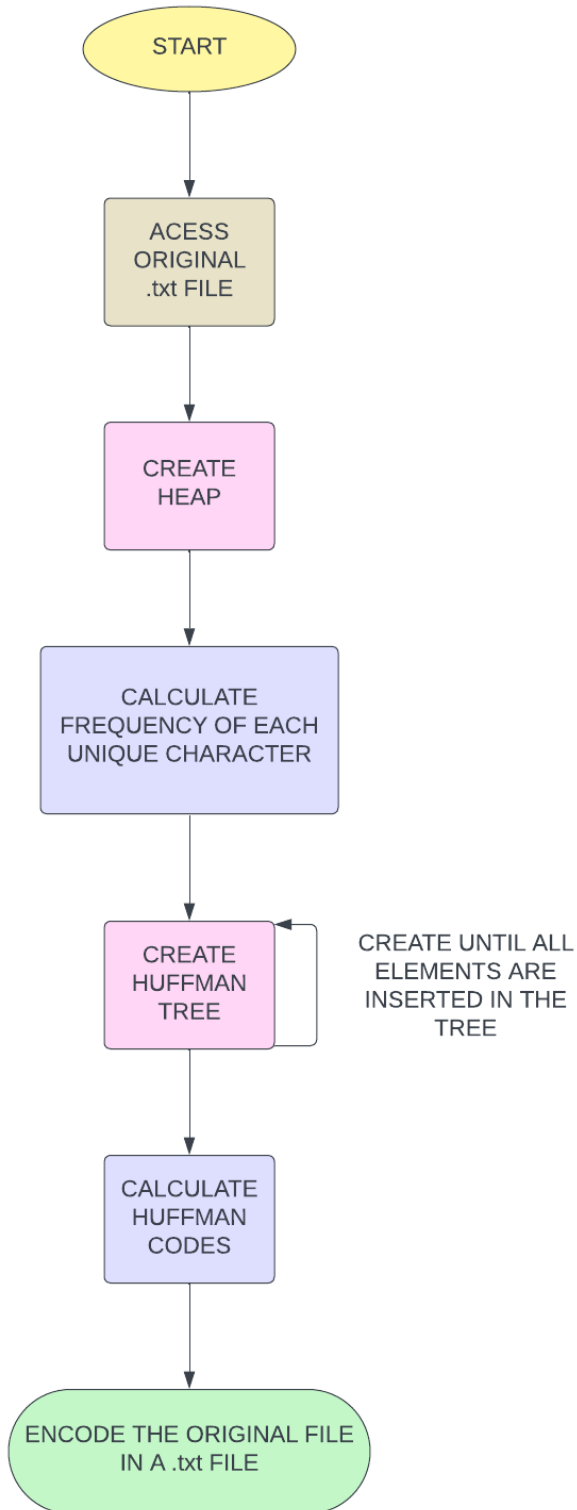
**Step 9:** Take input as a binary file.

**Step 10:** Translate the code according to the Huffman tree.

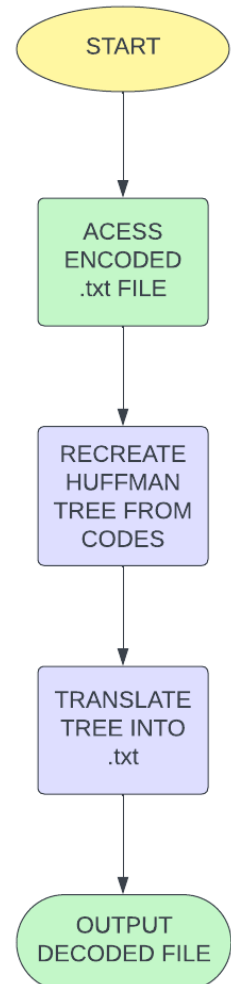
**Step 11:** Recreate the text file from the decoded characters.

## Flowchart

### Encoding



### Decoding



## Compression Ratio:


Original File(in KB)	Compressed File(in KB)	Compression Ratio
57	31	0.543859649
227	122	0.537444934
907	484	0.533627342
1814	966	0.532524807
3628	1931	0.532249173
	<b>Average Ratio:</b>	<b>0.535941181</b>

The compression ratio is calculated by the formula:


$$\text{Compression Ratio} = \text{Compressed File} / \text{Original File}$$

The average compression ratio for our Huffman Coding Project is **0.535941181**.

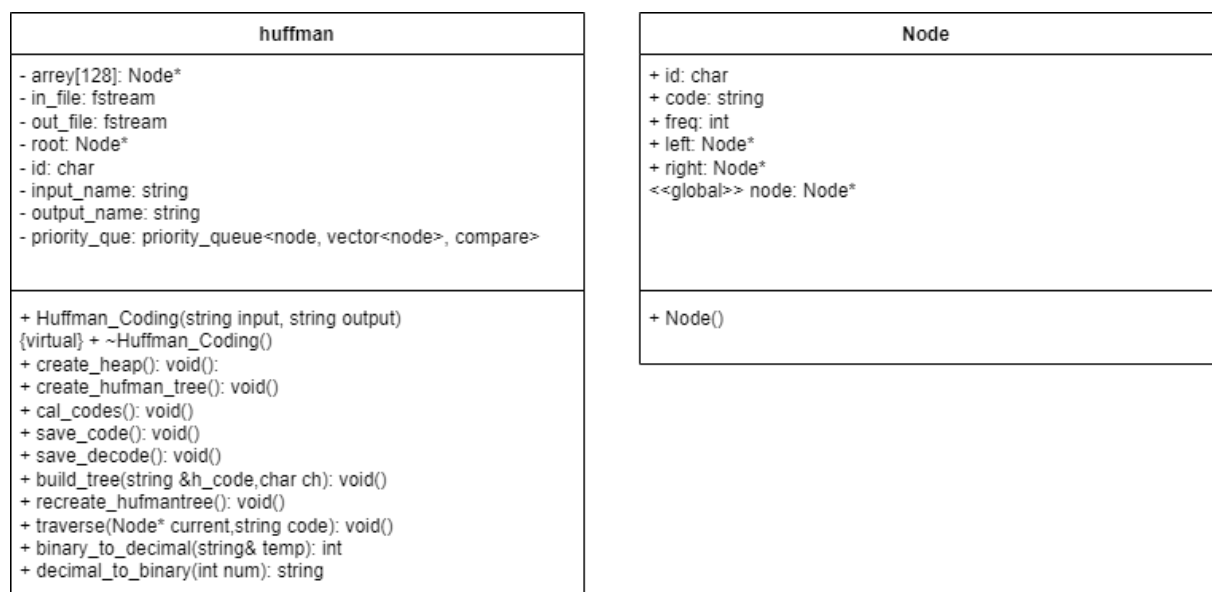
## Original File:

 Original	31/12/2023 12:13 am	Text Document	3,628 KB
---	---------------------	---------------	----------

## Encoded File:

 Encoded	31/12/2023 12:13 am	Text Document	1,931 KB
---	---------------------	---------------	----------

## UML Diagram:



## Conclusion:

Our project on Huffman Coding, a journey into data compression using binary trees and priority queues, has been a rewarding exploration. Beginning with project selection, we delved into the intricacies of Huffman trees and priority queues, implementing an efficient algorithm for text file compression and decompression.

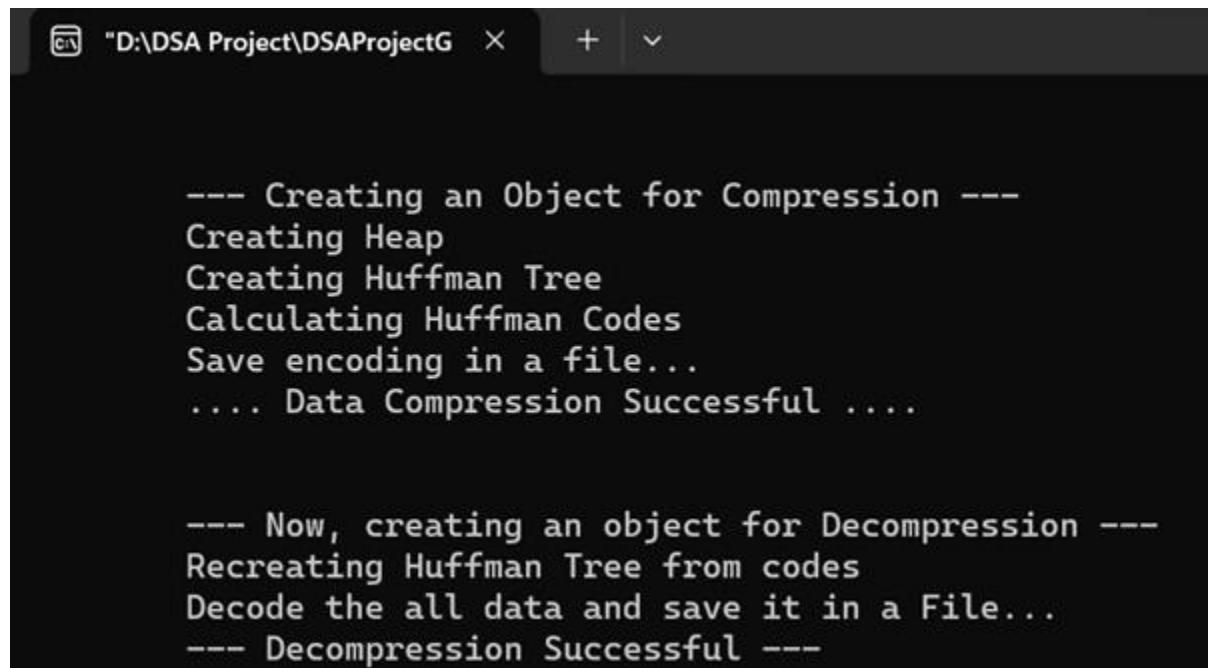
The project timeline, starting from December 1, 2023, witnessed phases of understanding, algorithm design, coding, and testing. We learned the significance of data structures like binary trees and priority queues, crucial for the success of Huffman Coding.

Our implementation allows users to compress and decompress text files, offering a practical solution for data compression. The importance of proper file path handling and considerations regarding character types in the original file were highlighted.

Huffman Coding's time complexity of  $O(n \log n)$  positions it as a valuable tool for text and image compression. Testing with various file sizes validated the algorithm's effectiveness, with the compression ratio serving as a performance metric.

In reflection, our project not only enhanced our coding skills but also deepened our understanding of algorithmic complexities and data structures. This journey has been both challenging and rewarding, contributing significantly to our academic growth.

## Output Of Program:



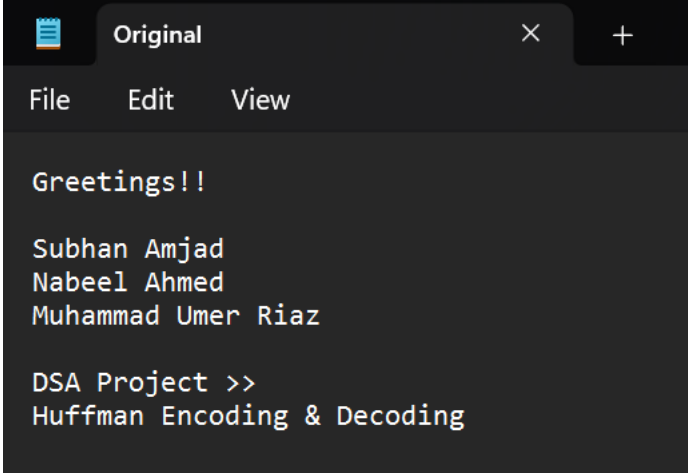
```
"D:\DSA Project\DSAProjectG  ×  +  ▾

--- Creating an Object for Compression ---
Creating Heap
Creating Huffman Tree
Calculating Huffman Codes
Save encoding in a file...
.... Data Compression Successful ....

--- Now, creating an object for Decompression ---
Recreating Huffman Tree from codes
Decode the all data and save it in a File...
--- Decompression Successful ---
```

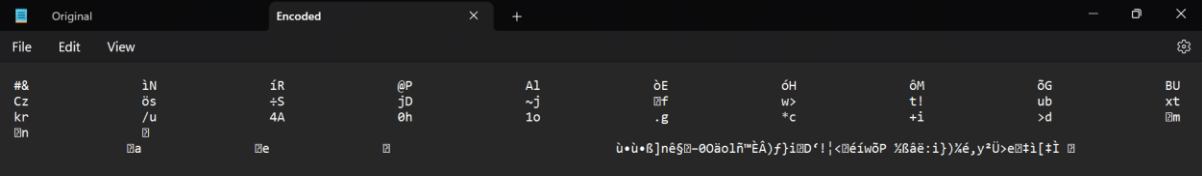


### Input File # 1:



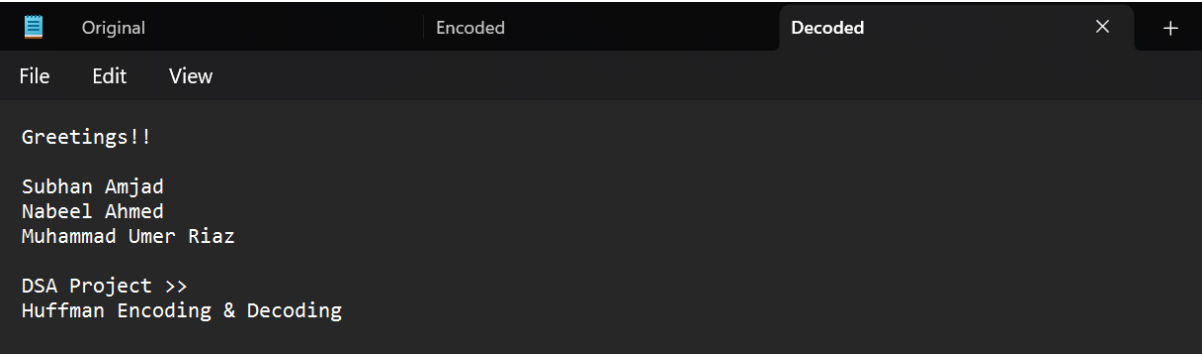
```
Greetings!!  
  
Subhan Amjad  
Nabeel Ahmed  
Muhammad Umer Riaz  
  
DSA Project >>  
Huffman Encoding & Decoding
```

### Encoded File # 1:



```
#&    iN    iR    @P    A1    òE    óH    òM    òG    BU  
Cz    os    +S    jD    ~j    W>    t!    ub    xt  
kP    /u    4A    0h    lo    -g    *c    +i    >d    0m  
0n  
0a    0e  
ù•ù•0]në§0-00äolñ~ÈÄ)f}i00'!|<0éiw0P %Bäe:i}}Xé,y²U>e0+i[+i 0
```

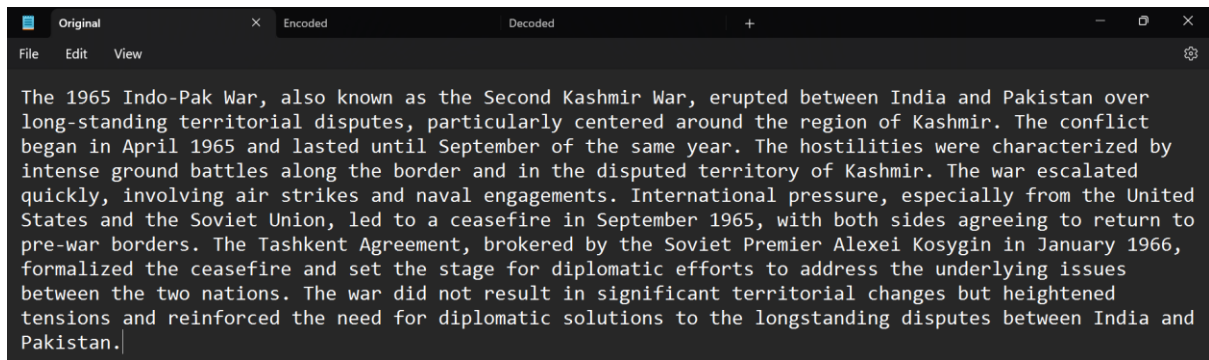
### Decoded File # 1:



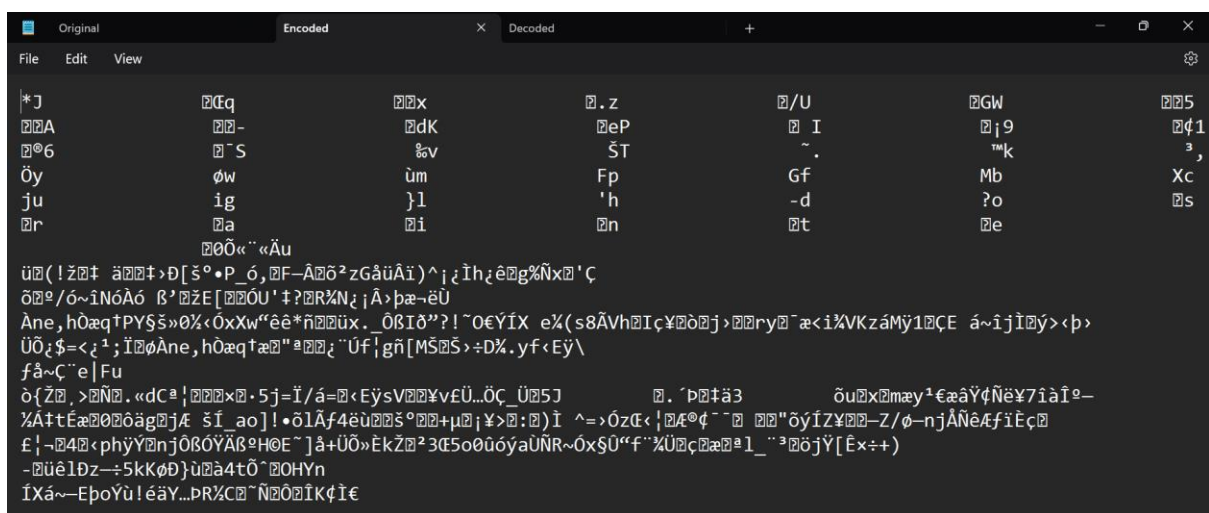
```
Greetings!!  
  
Subhan Amjad  
Nabeel Ahmed  
Muhammad Umer Riaz  
  
DSA Project >>  
Huffman Encoding & Decoding
```

**Original File same as Decoded File**

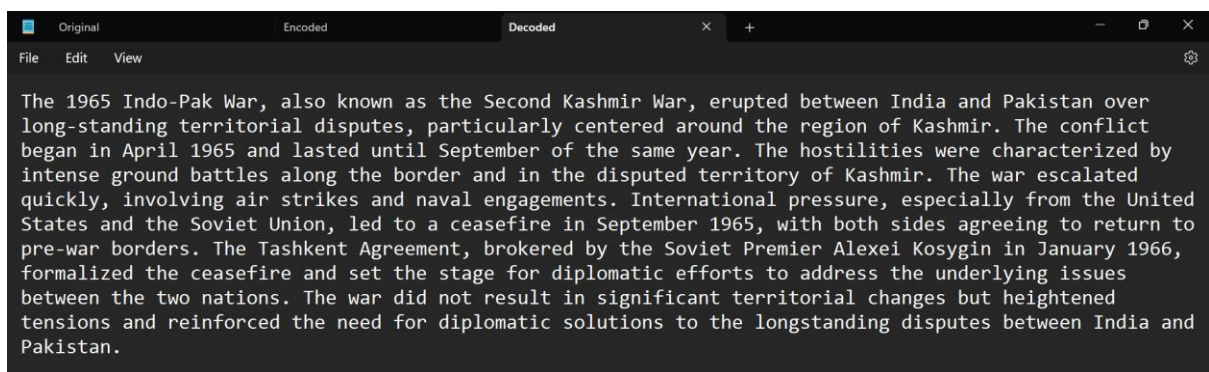
## Input File # 2:



## Encoded File # 2:



## Decoded File # 2:



**Original File same as Decoded File**