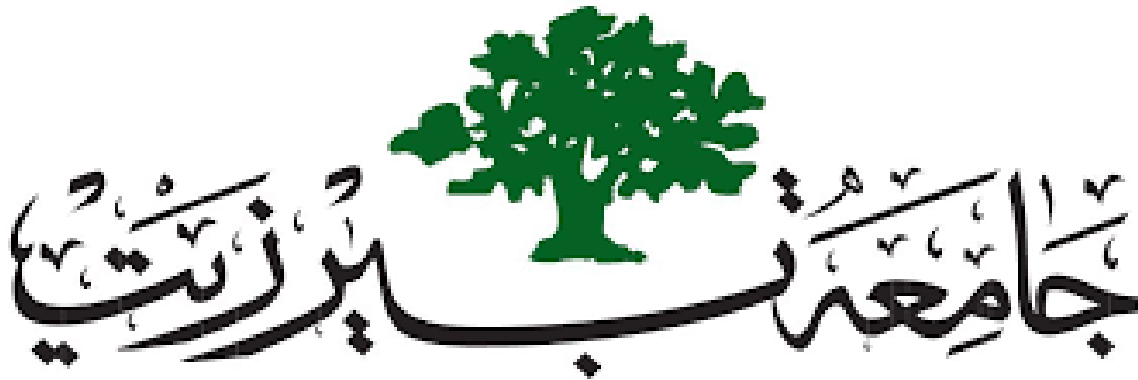


"بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ"



BIRZEIT UNIVERSITY

Faculty of Engineering and Technology.

Department of Electrical and Computer Engineering.

INFORMATION AND CODING THEORY - ENEE5304.

Course assignment: Huffman Code.

Students name (Prepared by): Razan Abdelrahman –1200531.

Maisam Alaa - 1200650.

Instructor: Dr. Wael Hashlamoun.

Section: 1.

Date: 10th | January | 2025.

Abstract.

This report explains the steps involved in applying Huffman coding, a lossless data compression method, to the text of *"To Build a Fire" by Jack London*. The process begins by calculating the frequency of each character in the text, followed by determining their probabilities. The entropy of the text is then computed to establish a theoretical lower bound for compression. Using a priority queue, a Huffman tree is built, and binary codes are assigned to characters based on their frequency. Finally, a comparison between Huffman encoding and ASCII encoding is performed to evaluate the efficiency of the compression method.

Table of contents.

Abstract.....	I
1. Introduction.....	1
2. Theoretical background.	2
2.1 Huffman coding steps.	2
2.2 Entropy Calculation.....	3
2.3 Average Number of Bits/Character.....	3
3. Simulation and results.	4
3.1 Character Analysis and Huffman Code words.	4
3.2 Entropy Calculation.....	5
3.3 Number of Bits Using ASCII (NASCII).....	5
3.4 Average Bits per Character Using Huffman.	5
3.5 Total Bits Using Huffman (Nhuffman).....	6
3.6 Compression Percentage.....	6
3.7 Subset Analysis.	7
4. Conclusion.	8
5. References.....	9
6. Appendix.....	10

List of figures.

Figure 3-1 Total number of characters in the story.	4
Figure 3-2 Frequencies, probabilities, and Huffman codes for characters.	4
Figure 3-3 Entropy value.	5
Figure 3-4 Number of bits needed using ASCII.....	5
Figure 3-5 Comparison between Huffman average bits and Entropy.....	5
Figure 3-6 Total number of bits in Huffman code.....	6
Figure 3-7 Percentage value.	6
Figure 3-8 Subset analysis for selected characters.	7

1. Introduction.

A Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code is Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. (Doctor of Science). Student at MIT, and published in the 1952 paper "*A Method for the Construction of Minimum-Redundancy Codes*" [1].

The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream [2].

2. Theoretical background.

2.1 Huffman coding steps.

Huffman coding is a lossless data compression method that assigns shorter codes to more frequently occurring characters and longer codes to less frequent ones. It uses a binary tree structure to achieve this and ensures the prefix property, where no code is a prefix of another [3].

Step 1: Character Frequency Calculation.

- The text of "*To Build a Fire*" by Jack London" was analyzed to count the occurrences of each character.
- These counts were divided by the total number of characters to calculate the probability of each character.

$$P_i = \frac{f_i}{N}$$

Where:

- P_i : The probability of the i-th character.
- f_i : The frequency of the i-th character (how many times it appears in the text).
- N : The total number of characters in the text.

Step 2: Sort Symbols.

Arrange the symbols in descending order of their frequencies.

Step 3: Merge Nodes.

Identify the two symbols with the lowest frequencies and merge them into a new node. The frequency of this new node is calculated as, where and are the frequencies of the two nodes being merged.

Step 4: Assign Binary Digits.

Assign a binary digit (e.g., 0 and 1) to each branch of the new node. Append these binary digits to the original symbols to form the Huffman code words.

Step 5: Repeat Merging.

Continue merging the two nodes with the smallest frequencies until only one root node remains, representing the complete Huffman tree.

Step 6: Assign Codes to Symbols.

Traverse the tree to assign binary codes to each character. Moving left adds 0, and moving right adds 1.

2.2 Entropy Calculation.

In information theory, an entropy coding (or entropy encoding) is any lossless data compression method that attempts to approach the lower bound declared by Shannon's source coding theorem [4], was computed using the formula:

$$H(X) = - \sum_{i=1}^n P_i \log_2 P_i$$

Where:

- $H(X)$ is the entropy.
- P_i is the probability of the i-th character.
- n is the total number of unique characters in the text.

2.3 Average Number of Bits/Character.

The average number of bits per character in the Huffman-encoded text is given by:

$$L = \sum_{i=1}^n P_i \times L_i$$

Where:

- P_i is the probability of the i-th character.
- L_i is the length of the Huffman code for the i-th character.

3. Simulation and results.

3.1 Character Analysis and Huffman Code words.

The code in the appendix (*Appendix: huffman_encoding.py*) uses the *calculate_frequencies* function to count how often each character appears in the story. The *calculate_probabilities* function then calculates the likelihood of each character based on its frequency. The program also calculates the total number of characters in the story, which is shown in Figure 3.1.

```
-----  
Total number of characters in the story is: 37705  
-----
```

Figure 3-1 Total number of characters in the story.

To generate the Huffman codes, the *build_huffman_tree* function creates a binary tree using the character frequencies. The *generate_huffman_codes* function then assigns binary codes to each character by traversing the tree. These codes, along with their corresponding frequencies and probabilities, are displayed in Figure 3.2, showing how each character is efficiently encoded.

Symbol	Frequency	Probability	Huffman Code	Code Length
' '	7048	0.18692	111	3
','	3	0.00008	00011111011011	14
''''	2	0.00005	00011111011001	14
'''''	20	0.00053	00011111001	11
'.'	436	0.01156	000110	6
'-'	89	0.00236	000111111	9
'_'	414	0.01098	000100	6
'>'	2	0.00005	00011111011010	14
'<'	26	0.00069	0001111000	10
'?'	1	0.00003	00011111011000	14
'a'	2264	0.06005	1001	4
'b'	484	0.01284	100000	6
'c'	779	0.02066	110110	6
'd'	1515	0.04018	11010	5
'e'	3887	0.10309	010	3
'f'	794	0.02106	00000	5
'g'	620	0.01644	100001	6
'h'	2278	0.06042	1010	4
'i'	1983	0.05259	0110	4
'j'	20	0.00053	00011111010	11
'k'	304	0.00806	1011000	7
'l'	1127	0.02989	10001	5
'm'	678	0.01798	101101	6
'n'	2077	0.05509	0111	4
'o'	1971	0.05227	0011	4
'p'	421	0.01117	000101	6
'q'	17	0.00045	00011111000	11
'r'	1481	0.03928	10111	5
's'	1795	0.04761	0010	4
't'	2937	0.07789	1100	4
'u'	800	0.02122	00001	5
'v'	179	0.00475	0001110	7
'w'	788	0.02090	110111	6
'x'	34	0.00090	0001111001	10
'y'	356	0.00944	1011001	7
'z'	61	0.00162	000111101	9
'_'	14	0.00037	000111110111	12

Figure 3-2 Frequencies, probabilities, and Huffman codes for characters.

Characters that appear more frequently, like 'o' are assigned shorter binary codes, while less frequent characters, such as '?' are assigned longer codes. This approach ensures that the text is compressed efficiently by reducing the total number of bits needed for storage

3.2 Entropy Calculation.

The code in the appendix (*Appendix: huffman_encoding.py*) uses the *calculate_entropy* function to compute the entropy of the text. This function takes the probabilities of each character and determines the theoretical lower bound for the average number of bits required to encode the text.

The entropy of the text was computed and is shown in Figure 3.3. This value represents the minimum average number of bits needed to encode the text based on the character probabilities.

```
-----  
Entropy of the alphabet: 4.172049 bits/character  
-----
```

Figure 3-3 Entropy value.

3.3 Number of Bits Using ASCII (NASCII).

The code in the appendix (*Appendix: huffman_encoding.py*) uses the *calculate_nascii* function to compute the total number of bits required to encode the text using ASCII encoding. This function multiplies the total number of characters in the text by 8, as ASCII uses a fixed-length 8-bit code for every character regardless of its frequency.

The calculated total number of bits using ASCII encoding is displayed in Figure 3.4.

```
-----  
Number of bits needed using ASCII (NASCII): 301640 bits  
-----
```

Figure 3-4 Number of bits needed using ASCII.

3.4 Average Bits per Character Using Huffman.

The code in the appendix (*Appendix: huffman_encoding.py*) uses the *calculate_average_bits* function to determine the average number of bits required to encode a character using Huffman coding. This function analyzes the probabilities of each character and the lengths of their assigned Huffman codes to compute the average.

The calculated average number of bits per character using Huffman encoding is displayed in Figure 3.5.

```
-----  
Entropy of the alphabet: 4.172049 bits/character  
Average number of bits/character using Huffman code: 4.218539 bits/character  
Comparison: Huffman Avg Bits vs Entropy -> 4.218539 vs 4.172049  
-----
```

Figure 3-5 Comparison between Huffman average bits and Entropy.

By comparing the entropy with the actual average number of bits per character achieved by Huffman encoding, we can evaluate how close the encoding is to the theoretical limit.

3.5 Total Bits Using Huffman (Nhuffman).

The code in the appendix (*Appendix: [huffman_encoding.py](#)*) uses the *calculate_nhuffman* function to compute the total number of bits required to encode the text using Huffman coding. This function multiplies the frequency of each character by the length of its corresponding Huffman code and sums the results for all characters.

The total number of bits required to encode the text using Huffman coding is displayed in Figure 3.6.

```
-----  
Total number of bits using Huffman code (Nhuffman): 159060 bits  
-----
```

Figure 3-6 Total number of bits in Huffman code.

Huffman coding significantly reduces the total number of bits needed compared to ASCII encoding. The reduction is achieved by assigning shorter codes to frequently occurring characters. This optimization directly translates into more efficient storage and transmission of data, as shown by the reduced total bit count compared to the fixed-length encoding of ASCII.

3.6 Compression Percentage.

The code in the appendix (*Appendix: [huffman_encoding.py](#)*) uses the *calculate_compression_percentage* function to determine how much compression was achieved. This function calculates the percentage reduction in bits by comparing the total bits required for ASCII encoding (NASCII) and Huffman encoding (Nhuffman).

The calculated compression percentage is shown in Figure 3.7.

```
-----  
Compression Percentage: 47.2683%  
-----
```

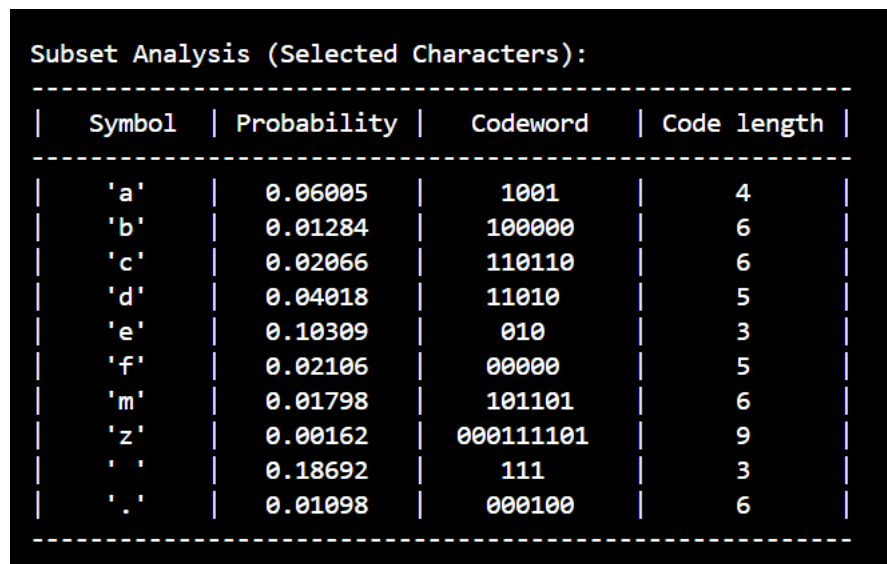
Figure 3-7 Percentage value.

The compression percentage shows the efficiency of Huffman coding in reducing storage requirements. A high compression percentage indicates that the text was encoded much more compactly than with ASCII. This result highlights the advantage of using variable-length encoding over fixed-length encoding, particularly for texts with uneven character frequencies.

3.7 Subset Analysis.

The code in the appendix (*Appendix: huffman_encoding.py*) uses the *print_subset_analysis* function to perform a focused analysis on a selected subset of characters. This function calculates and displays the probabilities, Huffman codes, and code lengths for specific characters chosen from the text.

The results of the subset analysis are displayed in Figure 3.8.



Symbol	Probability	Codeword	Code length
'a'	0.06005	1001	4
'b'	0.01284	100000	6
'c'	0.02066	110110	6
'd'	0.04018	11010	5
'e'	0.10309	010	3
'f'	0.02106	00000	5
'm'	0.01798	101101	6
'z'	0.00162	000111101	9
' '	0.18692	111	3
'.'	0.01098	000100	6

Figure 3-8 Subset analysis for selected characters.

The subset analysis shows that the adaptability of Huffman coding to different character frequencies. Frequently occurring characters in the subset are assigned shorter codes, while less common characters receive longer codes. This targeted analysis reaffirms the overall efficiency of the Huffman encoding process and provides a clearer understanding of its impact on specific parts of the text.

4. Conclusion.

This project showed how Huffman coding can compress text by assigning shorter codes to frequent characters and longer codes to less frequent ones. Using the story "*To Build a Fire*" by Jack London, the implementation reduced the total number of bits needed compared to ASCII encoding, which uses fixed-length codes.

The average number of bits per character in Huffman coding was very close to the entropy, showing that the method is near its theoretical best. The compression percentage confirmed that Huffman coding is much more efficient than ASCII.

The results from character analysis, entropy calculation, and subset evaluation proved that Huffman coding is a practical and effective method for data compression. It reduces storage space while maintaining the quality of the original data.

5. References.

- [1] [Online]. Available: https://en.wikipedia.org/wiki/Huffman_coding. [Accessed 27 12 2024].
- [2] [Online]. Available: <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>. [Accessed 26 12 2024].
- [3] [Online]. Available: <https://www.programiz.com/dsa/huffman-coding>. [Accessed 27 12 2024].
- [4] [Online]. Available:
https://en.wikipedia.org/wiki/Entropy_coding#:~:text=In%20information%20theory%2C%20an%20entropy,equal%20to%20the%20entropy%20of. [Accessed 27 12 2024].

6. Appendix.

Huffman_encoding.py:

```
from collections import Counter
import heapq
import docx2txt
import math

# Node class for the Huffman tree
class Node:
    def __init__(self, char=None, frequency=0, left=None, right=None):
        self.char = char
        self.frequency = frequency
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.frequency < other.frequency

# Step 1: Load the .docx file.
def load_docx(file_path):
    """Read content from a .docx file using docx2txt."""
    try:
        return docx2txt.process(file_path)
    except Exception as e:
        print(f"Error loading file: {e}")
        return ""

# Step 2: Preprocess the text.
def preprocess_text(text):
    """
    Convert text to lowercase and remove newline characters.
    All other symbols and characters are allowed.
    """
    # Convert to lowercase.
    text = text.lower()
    # Remove newline characters.
    text = text.replace("\n", "")
    return text

# Step 3: Analyze character frequencies.
def calculate_frequencies(text):
    """Count the frequency of each character in the text."""
    return Counter(text)

# Step 4: Calculate probabilities.
def calculate_probabilities(frequencies):
    """Calculate the probability of each character."""
```

```

    total_characters = sum(frequencies.values())
    probabilities = {char: freq / total_characters for char, freq in
frequencies.items()}
    return probabilities

# Step 5: Calculate entropy.
def calculate_entropy(probabilities):
    """Calculate the entropy of the alphabet."""
    entropy = -sum(p * math.log2(p) for p in probabilities.values() if p > 0)
    return entropy

# Step 6: Build Huffman tree.
def build_huffman_tree(frequencies):
    """Build the Huffman tree using a priority queue."""
    heap = [Node(char, freq) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(frequency=left.frequency + right.frequency, left=left,
right=right)
        heapq.heappush(heap, merged)

    return heap[0]

# Step 7: Generate Huffman codes.
def generate_huffman_codes(node, code="", codes=None):
    """Generate Huffman codes by traversing the tree."""
    if codes is None:
        codes = {}
    if node.char is not None:
        codes[node.char] = code
    if node.left:
        generate_huffman_codes(node.left, code + "0", codes)
    if node.right:
        generate_huffman_codes(node.right, code + "1", codes)
    return codes

# Step 8: Calculate NASCII.
def calculate_nascii(total_characters):
    """Calculate the number of bits needed to encode the text using ASCII."""
    return total_characters * 8

# Step 9: Calculate the average number of bits (bits/character).
def calculate_average_bits(probabilities, huffman_codes):
    """Calculate the average number of bits per character using Huffman code."""

```

```

        return sum(probabilities[char] * len(code) for char, code in
huffman_codes.items())

# Step 10: Calculate NHUFFMAN.
def calculate_nhuffman(frequencies, huffman_codes):
    """Calculate the total number of bits needed to encode the text using Huffman
code."""
    return sum(frequencies[char] * len(code) for char, code in
huffman_codes.items())

# Step 11: Calculate compression percentage.
def calculate_compression_percentage(nascii, nhuffman):
    """Calculate the percentage of compression accomplished by Huffman encoding."""
    return ((nascii - nhuffman) / nascii) * 100

# Print the frequency in tabular format.
def print_results(frequencies, probabilities, huffman_codes):
    """Print the frequencies, probabilities, and Huffman codes in alphabetical
order."""
    print("-----")
    print("
")
    print("
| Symbol | Frequency | Probability | Huffman Code | Code Length
|")
    print("-----")
    print("
")
    # Sort characters alphabetically.
    for char in sorted(frequencies.keys()):
        freq = frequencies[char]
        prob = probabilities[char]
        code = huffman_codes[char]
        print(f"
| {repr(char):^9} | {freq:^9} | {prob:^11.5f} | {code:^15} |
{len(code):^11} |")

    print("-----")
    print("
")

# Subset analysis for Selected Characters.
def print_subset_analysis(frequencies, probabilities, huffman_codes, subset):
    """Print analysis for selected characters in alphabetical order."""
    print("\nSubset Analysis (Selected Characters):")
    print("-----")
    print("
| Symbol | Probability | Codeword | Code length |")
    print("-----")
    for char in (subset):
        if char in frequencies:
            prob = probabilities[char]
            code = huffman_codes[char]
            print(f"
| {repr(char):^9} | {prob:^11.5f} | {code:^12} | {len(code):^11} |")

```



```

    print("-----")

# Display menu for the user.
def display_menu():
    """Display the menu options for the program."""
    print("\nChoose a step to calculate.")
    print("a. Display number of characters, their frequencies and probabilities, and
codewords using Huffman in the story.")
    print("b. Calculate and display Entropy.")
    print("1. Calculate number of Bits using ASCII (NASCII).")
    print("2. Calculate average number of Bits/Character using Huffman.")
    print("3. Calculate total bits using Huffman (Nhuffman).")
    print("4. Display compression percentage.")
    print("5. Show subset analysis.")
    print("6. Exit.")

# Main execution.
def main():
    # File path to the text file.
    file_path = "To_Build_A_Fire_by_Jack_London.docx"

    # Load and preprocess the text.
    text = preprocess_text(load_docx(file_path))
    frequencies = calculate_frequencies(text)
    probabilities = calculate_probabilities(frequencies)
    entropy = calculate_entropy(probabilities)

    # Build the Huffman tree and generate codes.
    huffman_tree = build_huffman_tree(frequencies)
    huffman_codes = generate_huffman_codes(huffman_tree)

    total_characters = sum(frequencies.values())
    nascii = calculate_nascii(total_characters)
    average_bits = calculate_average_bits(probabilities, huffman_codes)
    nhuffman = calculate_nhuffman(frequencies, huffman_codes)
    compression_percentage = calculate_compression_percentage(nascii, nhuffman)

    # Menu loop.
    while True:
        display_menu()
        choice = input("\nEnter your choice: ").strip().lower()

        if choice == "a":
            print('-----')
            print("Total number of characters in the story is:", total_characters)
            print('-----')
            print('\n')

```

```

        print_results(frequencies, probabilities, huffman_codes)

    elif choice == "b":
        print('-----')
        print("Entropy of the alphabet:", round(entropy, 6), "bits/character")
        print('-----')

    elif choice == "1":
        print('-----')
        print("Number of bits needed using ASCII (NASCII):", nascii, "bits")
        print('-----')

    elif choice == "2":
        print('-----')
        print("Entropy of the alphabet:", round(entropy, 6), "bits/character")
        print("Average number of bits/character using Huffman code:",
round(average_bits, 6), "bits/character")
        print("Comparison: Huffman Avg Bits vs Entropy ->", round(average_bits,
6), "vs", round(entropy, 6))
        print('-----')

    elif choice == "3":
        print('-----')
        print("Total number of bits using Huffman code (Nhuffman):", nhuffman ,
"bits")
        print('-----')

    elif choice == "4":
        print('-----')
        print(f"Compression Percentage: {compression_percentage:.4f}%")
        print('-----')

    elif choice == "5":
        subset = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'z', ' ', '.']
        print_subset_analysis(frequencies, probabilities, huffman_codes, subset)

    elif choice == "6":
        print("Thank you!")
        break

    else:
        print("Invalid choice. Please try again.")

if __name__ == "__main__":

```

```
main()
```