THEORITICAL PERFORMANCE ANALYSIS

TIME COMPLEXITY ANALYSIS:

Time complexity refers to the total number of operations necessary to execute an algorithm on a large dataset. Furthermore, since each operation necessitates a certain amount of time for the computer to process, the count of operations can be interpreted as a measure of time. Our program is divided in two major segments –

1. ***Data Compression:*** Uses Hufman and RLE (Run-Length Encoding)

   - Huffman uses **MIN HEAP** data structure for inserting and extracting data from the tree. For every unique it must traverse the height of tree. So, for **n** number of values, time complexity becomes $\boldsymbol{O(nlog_2n)}$**.**

   - For a k length Huffman string, RLE processed each character only once. So, time complexity becomes $\boldsymbol{O(k)}$**.**

2. ***Decompression (Storing & Retrieving):***

   - For retrieving each character from the Prefix trie (Decoding RLE) it must traverse only once. So, for a **k** length of string, time complexity $\boldsymbol{O(k)}$**.**
   - Similarly, decoding **n** length of Huffman string from the tree it takes $\boldsymbol{O(n)}$ of time complexity.

Besides, there are several conditions and statements that takes constant amount of time $\left(\boldsymbol{O(1)}\right)$ for execution. Therefore, the total time complexity of the system becomes $\boldsymbol{O(nlog_2n)}$, which is the dominating factor of the program.

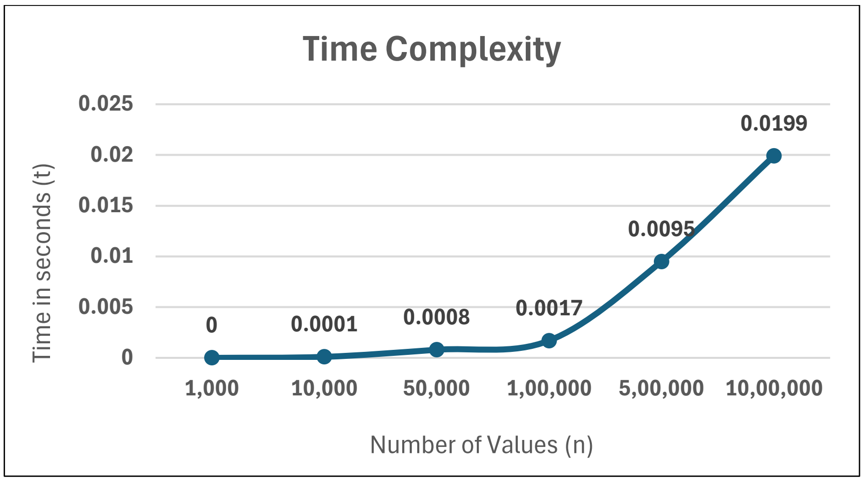
$$\textbf{Time (seconds)} = \boldsymbol{n \times \log_2 n \times t_{op}}$$

Here, $\boldsymbol{t_{op}}$ denotes the time of execution per operation in modern CPU. Preferably $\boldsymbol{1\ nanosecond = 10^{-9}}$

1. Data table:

Serial No.	Number of Values (n)	Equation $n \times \log_2 n \times t_{op}$	Time in seconds (t)
1	1,000	$1000 \times \log_2(1000) \times 10^{-9}$	0.0000
2	10,000	$10000 \times \log_2(10000) \times 10^{-9}$	0.0001
3	50,000	$50000 \times \log_2(50000) \times 10^{-9}$	0.0008
4	1,00,000	$100000 \times \log_2(100000) \times 10^{-9}$	0.0017
5	5,00,000	$500000 \times \log_2(500000) \times 10^{-9}$	0.0095
6	10,00,000	$1000000 \times \log_2(1000000) \times 10^{-9}$	0.0199

2. Line Graph:



Analysis: The line chart depicts a clear upward trajectory in time (t) as the number of values (n) increases. Initially, the progression of time is relatively slow; however, once the number of values surpasses 50,000, the rate of time increase becomes significantly more evident. This relationship suggests a non-linear correlation between time and the number of values, indicating that performance remains effective for larger datasets, despite the continuous rise in time as n expands.

SPACE COMPLEXITY ANALYSIS:

The execution of an algorithm on a computer requires a specific allocation of memory. The space complexity of a program quantifies the memory utilized during its execution. This complexity arises from the need for memory to accommodate both the input data and temporary variables throughout the program's runtime, encompassing both auxiliary and input space.

Similar to time complexity, space complexity is also derived from the two segments of the program compression and decompression. Here, is a detailed space analysis for our program.

1. Huffman Coding:

- **Input Text:** Since the input text of length **n** is stored in memory, space complexity is **O(n)**.
- **Frequency Table:** Stores the frequency of each unique character. If **m** is the number of unique characters, then space complexity is **O(m)**.
- **Huffman Tree:** Each unique character is represented by a leaf node, where **m** is the number of leaf nodes and has **m-1** internal nodes. Therefore, space complexity is **O(m).**
- **Codebook (Character to Binary Mapping):** Each character is mapped to a binary code of length up to $\mathbf{O(\log_2 m)}$. To accommodate the codes for all m characters, this specific space is required.
- **Compressed Output:** The length of the compressed binary string is directly related to the original text, which results in a space complexity of **O(n)**.

2. Run-Length Encoding (RLE):

- **RLE Compressed String:** In the worst case (no consecutive repeats), space complexity becomes **O(n)**.

3. Trie Data Structure:

- **Trie Nodes:** If the number of unique characters is **k** then space complexity is **O(k)**.

- **Stored Strings in Trie:** The RLE strings stored in the Trie contribute to the space used, and in the worst case, this can be **O(n)**.

$$\textbf{Overall Space Complexity} = \boldsymbol{O}(\boldsymbol{n} + \boldsymbol{m\log m} + \boldsymbol{k})$$

Where:

- **n** is length of the input text.
- **m** is the number of unique characters.
- **k** is the number of Trie nodes (depends on the RLE pattern, worst case **O(n)**)

In typical scenarios, **k** and **m** are much smaller than **n**. Therefore, the effective space complexity is $\boldsymbol{O}(\boldsymbol{n})$. To determine the space complexity of the program (measured in bytes), we consider the following components.

1. ***Original Text (text):*** n bytes.
2. ***Frequency Array (freq):*** 256 integers (each 4 bytes) = 1024 bytes.
3. ***Character Array (character):*** 256 bytes.
4. ***Huffman Tree:***
   - Each node is 21 bytes (1 byte for char, 4 bytes for frequency, 16 bytes for pointers).
   - Total size: (2 * size - 1) * 21 bytes (with size being up to 256).
5. ***Encoded Text (encodedText):*** n bytes.
6. ***RLE Encoded Text (rleEncoded):*** Worst-case size: 2 * n bytes.
7. ***Trie for RLE and Huffman Code Mapping:***
   - **Child Pointers**: Each node in the Trie can have up to 10 child pointers. And each pointer is 8 bytes. So, 10 child pointers take: $10 \times 8 = 80\ bytes$.
   - **Pointer to Huffman Code**: Each node also has a pointer to the corresponding Huffman code. This pointer is 8 bytes.

   Therefore, each node is 88 bytes, with 2 * n nodes in the worst case.
8. ***Decoded Huffman Text (decodedHuff):*** n bytes.
9. ***Final Decompressed Text (originalText):*** n bytes.
10. ***Other Variables:*** A small constant overhead C.

Total Space Formula:

$$\boldsymbol{S(n) = n + 1024 + 256 + (2 * 256 - 1) * 21 + n + 2n + (2n * 88) + n + n + C}$$

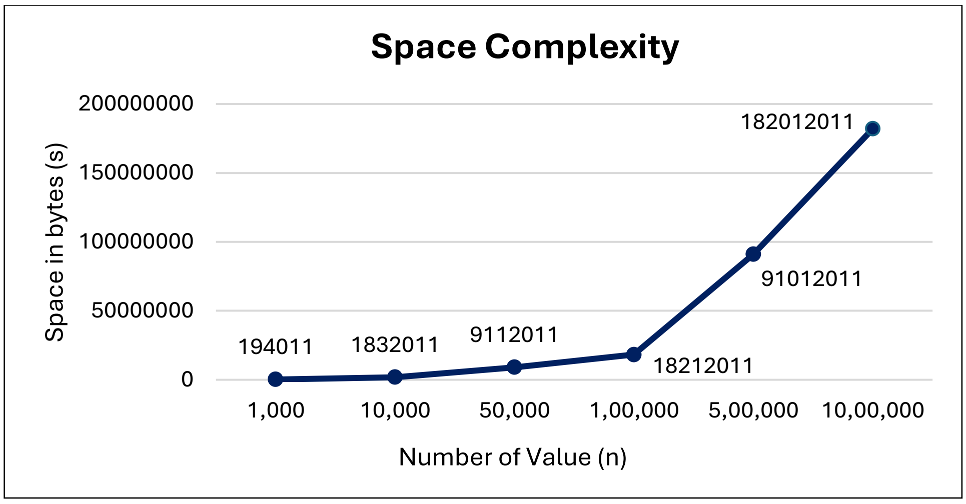
$\Rightarrow \boldsymbol{S(n) = n + 1024 + 256 + 10731 + n + 2n + 176n + n + n + C}$

$\Rightarrow \boldsymbol{S(n) = (182n) + 12011 + C\ (O(n)\ complexity)}$

1. *Data table:*

Serial No.	Number of Values (n)	Equation $S(n) = (182n) + 12011 + C$	Space in bytes (s)
1	1,000	$S(1000) = (182 \times 1000) + 12011 + C$	194011
2	10,000	$S(10000) = (182 \times 10000) + 12011 + C$	1832011
3	50,000	$S(50000) = (182 \times 50000) + 12011 + C$	9112011
4	1,00,000	$S(100000) = (182 \times 100000) + 12011 + C$	18212011
5	5,00,000	$S(500000) = (182 \times 500000) + 12011 + C$	91012011
6	10,00,000	$S(1000000) = (182 \times 1000000) + 12011 + C$	182012011

2. *Line Graph:*



Analysis: The theoretical analysis of space utilization in bytes reveals a steady and foreseeable increase corresponding to the growth in the number of values (n). This linear correlation suggests that the space demands expand in direct proportion to the size of the input, underscoring the effectiveness of space management for larger datasets as per the theoretical framework.

CODE

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

// Huffman Tree structure
struct TreeNode
{
   char character;
   unsigned freq;
   struct TreeNode *left, *right;
};

// MinHeap structure
struct Heap
{
   unsigned size;
   unsigned capacity;
   struct TreeNode **array;
};

// decompression structure (prefix)
struct Trie
{
   char *huffCode; // Store the Huffman code at the leaf
   struct Trie *children[10];
};

// Huffman Coding Functions
struct TreeNode *createTreeNode(char character, unsigned freq)
{
   struct TreeNode *node = (struct TreeNode *)malloc(sizeof(struct TreeNode));
   if (node==NULL)
   {
      fprintf(stderr, "Failed to allocate memory for TreeNode");
      exit(EXIT_FAILURE);
   }
   node->character = character;
   node->freq = freq;
   node->left = node->right = NULL;
   return node;
}

//Creates an empty heap to store huffman code
struct Heap *createHeap(unsigned capacity)
{
   struct Heap *heap = (struct Heap *)malloc(sizeof(struct Heap));
   if (heap==NULL)
   {
      fprintf(stderr, "Failed to allocate memory for Heap");
      exit(EXIT_FAILURE);
   }
```

```
    heap->size = 0;
    heap->capacity = capacity;
    heap->array = (struct TreeNode **)malloc(heap->capacity * sizeof(struct TreeNode *));
    if (heap->array=='\0')
    {
        fprintf(stderr, "Failed to allocate memory for Heap array");
        exit(EXIT_FAILURE);
    }
    return heap;
}

void swapHeapNodes(struct TreeNode **a, struct TreeNode **b)
{
    struct TreeNode *temp = *a;
    *a = *b;
    *b = temp;
}

//Maintains min heap property
void heapify(struct Heap *heap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    if (left < heap->size && heap->array[left]->freq < heap->array[smallest]->freq)
        smallest = left;
    if (right < heap->size && heap->array[right]->freq < heap->array[smallest]->freq)
        smallest = right;
    if (smallest != idx)
    {
        swapHeapNodes(&heap->array[smallest], &heap->array[idx]);
        heapify(heap, smallest);
    }
}

struct TreeNode *extractMin(struct Heap *heap)
{
    if (heap->size == 0)
        return NULL;

    struct TreeNode *temp = heap->array[0];
    heap->array[0] = heap->array[heap->size - 1];
    --heap->size;
    heapify(heap, 0);
    return temp;
}

void insertHeap(struct Heap *heap, struct TreeNode *node)
{
    if (heap->size == heap->capacity)
    {
        heap->capacity = heap->capacity * 2;
        heap->array = (struct TreeNode **)realloc(heap->array, heap->capacity * sizeof(struct TreeNode *));
        if (heap->array=='\0')
        {
            fprintf(stderr, "Memory reallocation failed for Heap array.");
            exit(EXIT_FAILURE);
        }
    }
        ++heap->size;
    int i = heap->size - 1;
    while (i && node->freq < heap->array[(i - 1) / 2]->freq)
    {
        heap->array[i] = heap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    heap->array[i] = node;
}

//Organizes an array of nodes into a valid min-heap
void buildHeap(struct Heap *heap)
{
    int n = heap->size - 1;
    for (int i = (n - 1) / 2; i >= 0; --i)
        heapify(heap, i);
}

//Combines the functionality of createHeap and buildHeap
struct Heap *createAndBuildHeap(char character[], int freq[], int size)
{
    struct Heap *heap = createHeap(size);
    for (int i = 0; i < size; ++i)
        heap->array[i] = createTreeNode(character[i], freq[i]);

    heap->size = size;
    buildHeap(heap);
    return heap;
}

//Constructs Hufman Tree
struct TreeNode *buildHuffTree(char character[], int freq[], int size)
{
    struct TreeNode *left, *right, *top;
    struct Heap *heap = createAndBuildHeap(character, freq, size);

    while (heap->size != 1)//until we get a single node in the heap which indicates the root of hufftree
    {
        left = extractMin(heap);
        right = extractMin(heap);
        top = createTreeNode('\0', left->freq + right->freq); // '\0' indicates "no character"
        top->left = left;
        top->right = right;
        insertHeap(heap, top);
    }
    struct TreeNode *root = extractMin(heap);
    free(heap->array);
    free(heap);
    return root;
}

void generateHuffCodes(struct TreeNode *root, char codes[256][256], char currentCode[], int top)
{
    if (root->left)
    {
        currentCode[top] = '0';
        currentCode[top + 1] = '\0';
        generateHuffCodes(root->left, codes, currentCode, top + 1);
    }
    if (root->right)
    {
        currentCode[top] = '1';
        currentCode[top + 1] = '\0';
        generateHuffCodes(root->right, codes, currentCode, top + 1);
    }
    if ((root->left)==NULL && (root->right)==NULL)
    {
        strcpy(codes[(unsigned char)root->character], currentCode);
    }
}

// Frequency Calculation
void calculateFreq(const char text[], int freq[], char character[], int *size)
{
    int len = strlen(text);
    for (int i = 0; i < len; i++)
    {
        unsigned char ch = text[i];
        int j;
        for (j = 0; j < *size; j++)
        {
            if (character[j] == ch)
            {
                freq[j]++;
                break;
            }
        }
        if (j == *size)
        {
            character[*size] = ch;
            freq[*size] = 1;
            (*size)++;
        }
    }
}

// RLE Encoding
char* applyRLE(const char *encodedText)
{
    size_t len = strlen(encodedText);

    // Worst case scenario: no repeats, so RLE length = original length
    // Plus some extra space for counts (assuming counts are small)
    size_t buffer_size = len * 2 + 1;
    char *rleEncoded = (char *)malloc(buffer_size);
    if (rleEncoded==NULL)
    {
        fprintf(stderr, "Memory allocation failed for RLE encoded string.");
        exit(EXIT_FAILURE);
    }
    size_t j = 0;
    for (size_t i = 0; i < len; i++)
```

```
  {
    rleEncoded[j++] = encodedText[i];
    int count = 1;
    while (i + 1 < len && encodedText[i] == encodedText[i + 1])
    {
      count++;
      i++;
    }
    if (count > 1)
    {
      // Convert count to string
      char countStr[20];
      int countLen = sprintf(countStr, "%d", count);

      // Ensure buffer has enough space
      if (j + countLen >= buffer_size)
      {
        buffer_size = buffer_size * 2;

        //Retains existing data using realloc
        rleEncoded = (char *)realloc(rleEncoded, buffer_size);
        if (rleEncoded==NULL)
        {
          fprintf(stderr, "Memory reallocation failed for RLE encoded string.");
          exit(EXIT_FAILURE);
        }
      }
      memcpy(&rleEncoded[j], countStr, countLen);
      j = j + countLen; // Update j and append the count in the existing string using sprintf
    }
  }
  rleEncoded[j] = '\0';
  return rleEncoded;
}

// Trie Creation and Insert
struct Trie *createTrieNode()
{
  struct Trie *node = (struct Trie *)malloc(sizeof(struct Trie));
  if (node==NULL)
  {
    fprintf(stderr,  "Memory allocation failed for Trie node.\n");
    exit(EXIT_FAILURE);
  }
  node->huffCode = NULL;
  for (int i = 0; i < 10; ++i)
    node->children[i] = NULL;
  return node;
}

void insertTrie(struct Trie *root, const char *rleEncoded, const char *huffCode)
{
  struct Trie *current = root;
  for (int i = 0; rleEncoded[i] != '\0'; i++)
  {
    if (isdigit(rleEncoded[i])=='\0')
    {
```

```c
            fprintf(stderr,  "RLE encoding contains non-digit characters in position %d.\n", i);
            exit(EXIT_FAILURE);
        }
        int index = rleEncoded[i] - '0';
        if (index < 0 || index > 9)
        {
            fprintf(stderr,  "RLE encoding digit out of range (0-9) at position %d.\n", i);
            exit(EXIT_FAILURE);
        }
        if (current->children[index] == NULL)
        {
            current->children[index] = createTrieNode();
        }
        current = current->children[index];
    }
    current->huffCode = strdup(huffCode);
    if (current->huffCode==NULL)
    {
        fprintf(stderr, "Memory allocation failed for Huffman code in Trie.");
        exit(EXIT_FAILURE);
    }
}

// RLE Decoding
void decodeRLE(struct Trie *root, const char *rleEncodedText, char **decodedHuff)
{
    struct Trie *current = root;
    int i = 0;
    size_t decoded_size = strlen(rleEncodedText) * 8 + 1; // Approximate size
    *decodedHuff = (char *)malloc(decoded_size);
    if ((*decodedHuff)=='\0')
    {
        fprintf(stderr,  "Memory allocation failed for decoded Huffman string.\n");
        exit(EXIT_FAILURE);
    }
    (*decodedHuff)[0] = '\0';
    while (rleEncodedText[i] != '\0')
    {
        if (isdigit(rleEncodedText[i])=='\0')
        {
            fprintf(stderr,  "RLE encoded text contains non-digit characters at position %d.\n", i);
            free(*decodedHuff);
            exit(EXIT_FAILURE);
        }
        int index = rleEncodedText[i] - '0';
        if (index < 0 || index > 9)
        {
            fprintf(stderr,  "RLE encoding digit out of range (0-9) at position %d.\n", i);
            free(*decodedHuff);
            exit(EXIT_FAILURE);
        }
        if (current->children[index])
        {
            current = current->children[index];
            i++;
            if (current->huffCode)
            {
```

```
            size_t huff_len = strlen(current->huffCode);
            size_t current_len = strlen(*decodedHuff);
            if (current_len + huff_len + 1 > decoded_size)
            {
                decoded_size = decoded_size * 2;
                *decodedHuff = (char *)realloc(*decodedHuff, decoded_size);
                if ((*decodedHuff)==NULL)
                {
                    fprintf(stderr, "Memory reallocation failed for decoded Huffman string.\n");
                    exit(EXIT_FAILURE);
                }
            }
            strcat(*decodedHuff, current->huffCode);
            current = root;
        }
    }

    else//when there is only one character in the array
    {
        // If current path does not exist, it might be a count
        if (isdigit(rleEncodedText[i]))
        {
            int count = 0;
            while (isdigit(rleEncodedText[i]))
            {
                count = count * 10 + (rleEncodedText[i] - '0');
                i++;
            }
            if (current->huffCode == '\0')
            {
                fprintf(stderr, "Invalid RLE encoding: count without preceding Huffman code.\n");
                free(*decodedHuff);
                exit(EXIT_FAILURE);
            }
            size_t huff_len = strlen(current->huffCode);
            size_t current_len = strlen(*decodedHuff);
            size_t required_size = current_len + huff_len * count + 1;
            if (required_size > decoded_size)
            {
                while (required_size > decoded_size)
                {
                    decoded_size = decoded_size * 2;
                }
                *decodedHuff = (char *)realloc(*decodedHuff, decoded_size);
                if ((*decodedHuff)=='\0')
                {
                    fprintf(stderr, "Memory reallocation failed for decoded Huffman string.\n");
                    exit(EXIT_FAILURE);
                }
            }
            for (int k = 0; k < count; k++)
            {
                strcat(*decodedHuff, current->huffCode);
            }
            current = root;
        }
        else
```

```c
        {
          fprintf(stderr, "Invalid character in RLE encoded text at position %d.\n", i);
          free(*decodedHuff);
          exit(EXIT_FAILURE);
        }
      }
    }
}

// Huffman Decoding
void decodeHuff(struct TreeNode *root, const char *encodedText, char **originalText)
{
  struct TreeNode *current = root;
  int i = 0;
  size_t original_size = strlen(encodedText) + 1; // Max possible
  *originalText = (char *)malloc(original_size);
  if ((*originalText)==NULL)
  {
    fprintf(stderr, "Memory allocation failed for original text.\n");
    exit(EXIT_FAILURE);
  }
  (*originalText)[0] = '\0';
  size_t j = 0;
  while (encodedText[i] != '\0')
  {
    if (encodedText[i] == '0')
      current = current->left;
    else if (encodedText[i] == '1')
      current = current->right;
    else
    {
      fprintf(stderr, "Invalid character in Huffman encoded text at position %d.\n", i);
      free(*originalText);
      exit(EXIT_FAILURE);
    }
    if (current == NULL)
    {
      fprintf(stderr, "Traversal reached a NULL node at position %d.\n", i);
      free(*originalText);
      exit(EXIT_FAILURE);
    }
    if (current->left == NULL && current->right == NULL)
    {
      // Ensure there is enough space
      if (j + 1 >= original_size)
      {
        original_size = original_size *2;
        *originalText = (char *)realloc(*originalText, original_size);
        if ((*originalText)==NULL)
        {
          fprintf(stderr, "Memory reallocation failed for original text.\n");
          exit(EXIT_FAILURE);
        }
      }
      (*originalText)[j++] = current->character;
      (*originalText)[j] = '\0';
      current = root;
```

```c
        }
        i++;
    }
}

// Function to generate random input
char* generateRandomText(int length)
{
    constchar charset[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    const size_t charset_size = sizeof(charset) - 1;
    char *text = (char *)malloc((length + 1) * sizeof(char));
    if (text==NULL)
    {
        fprintf(stderr,  "Memory allocation failed for random text.\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < length; i++)
    {
        text[i] = charset[rand() % charset_size];
    }
    text[length] = '\0';
    return text;
}

// Function to print time and space information
void printTotalTimeAndSpace(clock_t start_time, clock_t end_time, size_t total_space)
{
    double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    printf("\nTime Taken: %f seconds\n", time_taken);
    printf("Total Space Consumed: %zu bytes\n", total_space);
}

int main()
{
    srand((unsigned int)time(NULL));  // random number generation
    int length;
    printf("Enter the length of random text to generate: ");
    scanf("%d", &length);

    // Generate random text
    char *text = generateRandomText(length);
    printf("Generated Random Text: %s\n", text);
    clock_t start_time = clock();

    int *freq = (int *)calloc(256, sizeof(int));
    if (freq==NULL)
    {
        fprintf(stderr,  "Memory allocation failed for frequency array.\n");
        free(text);
        return EXIT_FAILURE;
    }

    char *character = (char *)malloc(256 * sizeof(char));
    if (character==NULL)
    {
        fprintf(stderr,  "Memory allocation failed for character array.\n");
        free(text);
    free(freq);
    return EXIT_FAILURE;
  }

  int size = 0;
  calculateFreq(text, freq, character, &size);

  // Handling for single-character input
  if (size == 1)
  {
    char *rleEncoded = applyRLE(text);
    printf("Huffman Encoded String: 0\n");// Only one unique character, Huffman code is '0'
    printf("RLE Encoded String: %s\n", rleEncoded);
    printf("Original Text After Decompression: %s\n", text);
    free(text);
    free(rleEncoded);
    free(freq);
    free(character);
    return EXIT_SUCCESS;
  }

  struct TreeNode *root = buildHuffTree(character, freq, size);
  char codes[256][256] = {0};
  char currentCode[256];
  currentCode[0] = '\0';
  generateHuffCodes(root, codes, currentCode, 0);

  // Estimate the size of encodedText (each character can be up to 256 bits)
  size_t encoded_size = length * 16 + 1; // Adjust as needed
  char *encodedText = (char *)malloc(encoded_size);
  if (encodedText==NULL)
  {
    fprintf(stderr,  "Memory allocation failed for encoded text.\n");
    free(text);
    free(freq);
    free(character);

    return EXIT_FAILURE;
  }

  encodedText[0] = '\0';
  for (int i = 0; i < length; i++)
  {
    strcat(encodedText, codes[(unsigned char)text[i]]);
  }
  printf("\nHuffman Encoded String:\n%s\n", encodedText);

  char *rleEncoded = applyRLE(encodedText);
  printf("\nRLE Encoded String:\n%s\n", rleEncoded);

  struct Trie *trieRoot = createTrieNode();

  insertTrie(trieRoot, rleEncoded, encodedText);

  char *decodedHuff = NULL;
  decodeRLE(trieRoot, rleEncoded, &decodedHuff);
```

```
    char *originalText = NULL;
    decodeHuff(root, decodedHuff, &originalText);
    printf("\nOriginal Text After Decompression: %s\n", originalText);
    clock_t end_time = clock();

    // Estimate total space consumed
    size_t total_space =
        sizeof(struct TreeNode) * size +
        sizeof(struct Trie) +
        strlen(encodedText) +
        strlen(rleEncoded) +
        sizeof(char) * (length + 1) +
        sizeof(char) * (strlen(decodedHuff) + 1) +
        sizeof(char) * (strlen(originalText) + 1) +
        sizeof(char) * (256 + 256) + // freq and character arrays
        sizeof(struct Heap);
    printTotalTimeAndSpace(start_time, end_time, total_space);

    // Free allocated memory
    free(text);
    free(freq);
    free(character);
    free(encodedText);
    free(rleEncoded);
    free(decodedHuff);
    free(originalText);

    return EXIT_SUCCESS;
}
```

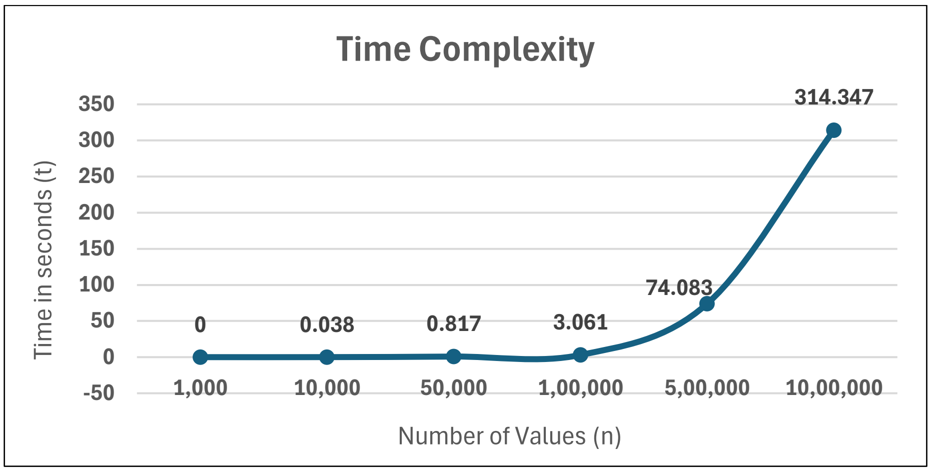
PRACTICAL PERFORMANCE ANALYSIS

TIME COMPLEXITY ANALYSIS:

1. ***Data table***: Collected from the code

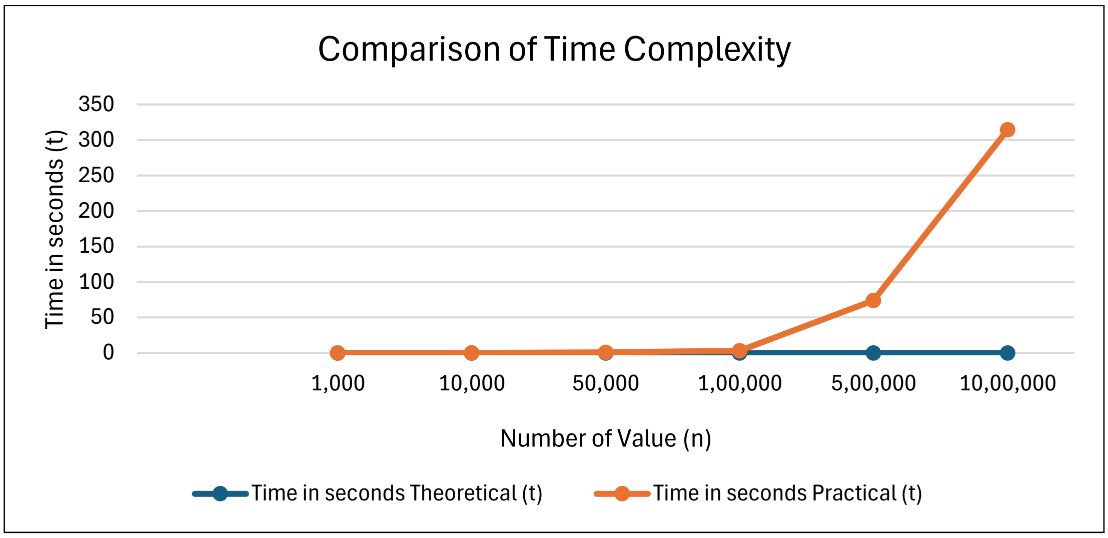
Serial No.	Number of Values (n)	Time in seconds (t)
1	1,000	0.0000
2	10,000	0.0380
3	50,000	0.8170
4	1,00,000	3.0610
5	5,00,000	74.0830
6	10,00,000	314.347

2. ***Line Graph:***



Analysis: The analysis reveals a notable increase in time (t) corresponding to an increase in the number of values (n). Initially, for smaller values of n, the rise in time is relatively gradual. However, once n exceeds 50,000, there is a marked acceleration in time. As n approaches 500,000 and beyond, the time experiences a substantial spike, indicating that the process significantly slows down with larger datasets, demonstrating a pattern akin to exponential growth.

COMPARISON:



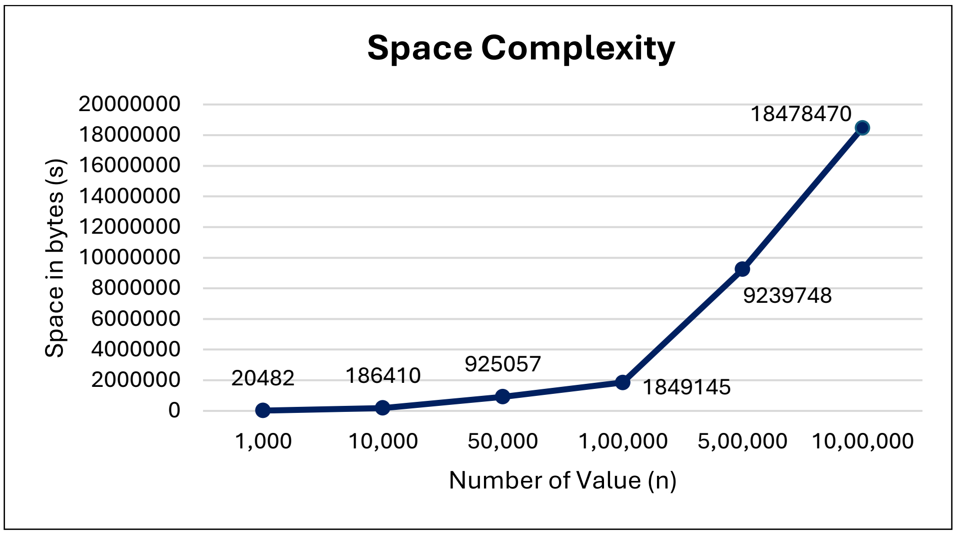
Analysis: The theoretical time shows a gradual and consistent rise, reflecting a nearly linear trend, while the practical time experiences a significant surge, especially with larger inputs. The growth pattern of practical time resembles that of exponential growth, diverging markedly from theoretical expectations as the size of the input increases. This suggests that inefficiencies exist within the practical implementation when managing larger datasets.

SPACE COMPLEXITY ANALYSIS:

1. ***Data table***: Collected from the code

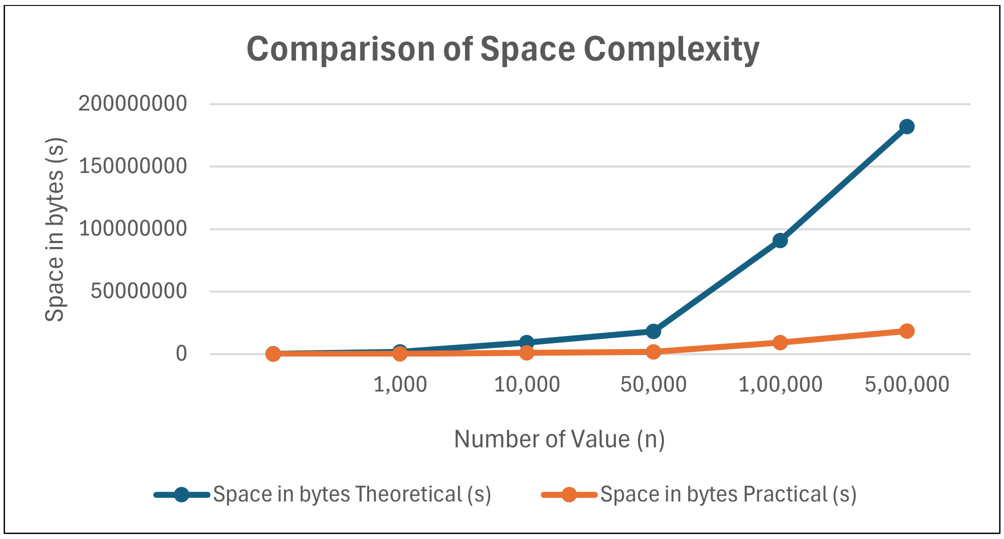
Serial No.	Number of Values (n)	Space in bytes (s)
1	1,000	20482
2	10,000	186410
3	50,000	925057
4	1,00,000	1849145
5	5,00,000	9239748
6	10,00,000	18478470

1. ***Line Graph:***



Analysis: The demand for storage space increases in a manner that is approximately linear relative to the quantity of values, as demonstrated by the observed trend of rising byte usage. Although the growth is not entirely proportional, it reveals a reliable pattern indicating that larger datasets necessitate considerably more storage capacity.

COMPARISON:



Analysis: The utilization of practical space consistently hovers around 10% of the theoretical space, demonstrating a notable efficiency in actual memory consumption when compared to theoretical projections. For example, when dealing with 1,000 values, the practical space utilized is 20,482 bytes, whereas the theoretical space amounts to 194,011 bytes. To determine the efficiency, $\frac{20{,}482}{194{,}011} \times 100\% = 10.56\%$ yields approximately 10%. This pattern is evident across all dataset sizes, indicating that the practical application effectively employs significantly less space than what theoretical estimates suggest. However, it should be noted that the theoretical calculations do not manage the ruse of several memory parts, which leads to disparity between the theoretical and practical values.

Tool Test: Tool vs. Market Alternative

In this part we are comparing our program with a real-life tool known as **G-zip**.

- G-zip uses algorithms DEFLATE (combination of Huffman and LZ77 which is similar to RLE) but gives better optimization for repetitive data. This software is widely used in Unix like operating system (Linux).

- Time Complexity: $\boldsymbol{O}(\boldsymbol{nlog_2n}) + \boldsymbol{O}(\boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{nlog_2n})$

- Space Complexity: $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{w})$

From the above we can clearly how this commercial tool is similar to ours. However, there are several reasons that performs better than our university level developed program.

Reasons why G-zip works better:

- Minimizes storage usage, so memory management is better.

- Sliding window technique handles efficiently large patterns. Instead of going through character by character, it uses a small group according to the window size. So, the decompression time is lot faster.

- Since it is commercially tested and developed, G-zip is better at handling errors and edge cases.

Potentiality in our Program:

- Dynamic data handling using trie and tree can give better performance where data is changing continuously. Example: Live Broadcast

- Since we use tree structure for storing and retrieving, we do not have to look for if any node is left behind while traversing.

- Common prefix, data are easily stored according to their common part, minimizing redundant storage.