Huffman Codes

Chapter 1: Introduction

1.1 Background

The background of the problem revolves around Huffman codes, which are a method for constructing efficient, lossless data compression algorithms, developed by David A. Huffman in 1953. Huffman's method involves creating a binary tree of nodes, which represents the frequency of occurrence for each data element, and assigning binary codes to each element such that shorter codes are given to more frequently occurring elements. This technique minimizes the total number of bits used in encoding the data, hence it's called minimum-redundancy codes.

The problem statement highlights a challenge with Huffman codes: they are not unique. Different Huffman trees can be created depending on the order of merging nodes with equal frequencies during the tree construction process, leading to different valid encoding schemes that achieve the same level of compression.

To illustrate, given the string "aaaxuaxz", you could encode it using multiple valid Huffman codes, as the frequencies of the characters vary. However, not all codes students might submit will be valid; some might not properly differentiate between characters, causing different strings to be encoded into the same binary sequence, which violates the fundamental principle of Huffman coding where each code must uniquely and unambiguously decode to its corresponding character (i.e., it must be a prefix code).

The main challenge posed by the problem is to verify whether a given set of Huffman codes (submitted by students) is correct. This involves checking if each code is indeed a prefix code, meaning no whole code is a prefix of any other. The task is to implement a computer program that can perform this verification efficiently, given a list of characters with their frequencies and proposed encoding schemes by students. The output should confirm whether each submitted set of codes is correct ("Yes") or not ("No").

1.2 Project Overview

Objective

Develop a program that verifies the correctness of Huffman codes submitted by students. Each submission needs to be evaluated to ensure it represents a valid prefix code that uniquely and accurately encodes a given set of characters based on their frequencies.

Key Components

1. **Data Input Handling**:
   - Parse the input to read the number of characters (N), their frequencies, and the student submissions.
   - Handle different character sets and frequency distributions.
2. **Prefix Code Validation**:
   - For each student submission, validate that the set of binary strings (codes) forms a valid prefix code.
   - Ensure no code is a prefix of another, which is essential for unique decodability.
3. **Efficiency Considerations**:
   - Given the constraints ($N \leq 63$, frequencies ≤ 1000, $M \leq 1000$ submissions), ensure that the program runs efficiently.
   - Use data structures like tries or sorted lists to efficiently check for prefix conditions.

Detailed Steps

1. **Input Processing**:
   - Read the number of distinct characters N.
   - For each character, read its frequency and store the data.
   - Read the number of student submissions M.
2. **Submission Validation**:
   - For each submission:
     - Read the N lines containing characters and their proposed codes.
     - Store the codes in a way that facilitates quick prefix checks, such as in a minheap.
     - Validate the codes to ensure that none is a prefix of another.
3. **Output Results**:
   - For each submission, output "Yes" if the set of codes forms a valid prefix code, otherwise output "No".

This project involves developing a robust solution that not only checks basic conditions but also efficiently handles potentially large datasets and ensures that all edge cases are covered, especially given the nuances of prefix codes in Huffman encoding.Chapter2: Algorithm Specification

2.1 Data Structure and Variables

Data Structures

1. **`HuffmanNode` Structure:**
   - **`weight`** : Integer representing the frequency of a character or sum of frequencies for internal nodes in the Huffman tree.
   - **`Left and Right`** : Pointers to the left and right child nodes, respectively, used to form the binary tree structure of the Huffman tree.
2. **`MinHeapNode` Structure:**
   - **`size`** : Integer tracking the current number of nodes in the min-heap.
   - **`data[70]`** : Array of pointers to `HuffmanNode` , representing the elements in the min-heap. It acts as a priority queue where the node with the lowest weight (highest priority) is at the root.

Variables

- `Char_num` : Integer representing the number of distinct characters to be encoded.
- `Test_num` : Integer indicating the number of student submissions to be validated.
- `f[70]` : Array storing the frequencies of each character.
- `c[70]` : Array storing the characters themselves.
- `sum` : Integer representing the total Weighted Path Length (WPL) of the constructed Huffman tree, used as a benchmark to validate student submissions.

```
int Char_num, Test_num, f[70], sum;
char c[70];

struct HuffmanNode{
    int weight;
    struct HuffmanNode* Left;
    struct HuffmanNode* Right;
};
struct MinHeapNode{
    int size;
    struct HuffmanNode* data[70];
};

/* Function Area */
struct MinHeapNode* CreateMinHeapNode(){
    struct MinHeapNode* MinHeap = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    MinHeap->size = 0;
    MinHeap->data[0] = (struct HuffmanNode*)malloc(sizeof(struct HuffmanNode));
    MinHeap->data[0]->weight = -1;
    MinHeap->data[0]->Left = NULL;
    MinHeap->data[0]->Right = NULL;
    // data[0] will never be used. start from data[1]
    return MinHeap;
}
struct HuffmanNode* CreateHuffmanNode(){
    struct HuffmanNode* Huffman = (struct HuffmanNode*)malloc(sizeof(struct HuffmanNode));
    Huffman->weight = 0;
    Huffman->Left = NULL;
    Huffman->Right = NULL;
    return Huffman;
}
```

2.2 Function: CreateMinHeapNode and CreateHuffmanNode

These functions initialize instances of `MinHeapNode` and `HuffmanNode` , respectively, setting default values and allocating memory.

```
struct MinHeapNode* CreateMinHeapNode(){
    struct MinHeapNode* MinHeap = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    MinHeap->size = 0;
    MinHeap->data[0] = (struct HuffmanNode*)malloc(sizeof(struct HuffmanNode));
    MinHeap->data[0]->weight = -1;
    MinHeap->data[0]->Left = NULL;
    MinHeap->data[0]->Right = NULL;
    // data[0] will never be used. start from data[1]
    return MinHeap;
}
struct HuffmanNode* CreateHuffmanNode(){
    struct HuffmanNode* Huffman = (struct HuffmanNode*)malloc(sizeof(struct HuffmanNode));
    Huffman->weight = 0;
    Huffman->Left = NULL;
    Huffman->Right = NULL;
    return Huffman;
}
```

2.3 Function: Insert

Inserts a new `HuffmanNode` into the min-heap (`MinHeapNode`), maintaining the heap property (smallest weights remain at the top).

```
void Insert(MinHeapNode* heap, HuffmanNode* node){
    heap->size++;
    int k = heap->size;
    while (node->weight < heap->data[k>>1]->weight){
        heap->data[k] = heap->data[k>>1];
        k = k>>1;
    }
    heap->data[k] = node;
    // node[k>>1] is node[k] 's father node
    // If the value to be inserted is greater than the value of the parent node,
    // then insert it into the current position
    // Else, continue to find the parent node and move the parent
    // node to the current node position.
}
```

2.4 Function: GetMinAndDelete

Removes and returns the node with the minimum weight from the heap, re-adjusting the heap to maintain its properties.

```
struct HuffmanNode* GetMinAndDelete(struct MinHeapNode* heap){
    struct HuffmanNode* Minterm = heap->data[1];
    struct HuffmanNode* temp = heap->data[heap->size --];
    // equals to insert temp into the new tree
    int parent = 1; // Start from the root node
    while(parent * 2 <= heap->size){
        int Leftchild = parent * 2;
        int Rightchild = parent * 2 + 1;
        int Smallerchild = Leftchild;
        if(Leftchild != heap->size){ // If the current node has a right child
            // If the left child is greater than the right child
            if(heap->data[Leftchild]->weight > heap->data[Rightchild]->weight)
                Smallerchild = Rightchild;  // Then the right child is the smaller child
        }
        // If the value to be inserted is less than the smaller child
        if(temp->weight <= heap->data[Smallerchild]->weight)
            break;  // Then the current node is in the correct position
        else // Otherwise, move the smaller child to the parent node
            heap->data[parent] = heap->data[Smallerchild];
        parent = Smallerchild;
    }
    heap->data[parent]=temp;
    return Minterm;
}
```

Function: buildHuffman

Constructs the Huffman tree by repeatedly removing the two nodes with the smallest weights, merging them into a new node, and inserting the new node back into the heap, until only one node remains (the root of the Huffman tree).

```
struct HuffmanNode* buildHuffman(struct MinHeapNode* heap){
    struct HuffmanNode* node;
    int times = heap->size;
    for(int i = 1;i < times;i ++){
        node = CreateHuffmanNode();
        node->Left = GetMinAndDelete(heap);
        node->Right = GetMinAndDelete(heap);
        node->weight = node->Left->weight + node->Right->weight;
        Insert(heap, node);
        // The number of nodes in the heap decreases by 1 each time
        // Thus, the loop will run (times-1) times
    }
    return GetMinAndDelete(heap);   // Return the root node of the Huffman tree
}
```

Function: WPL

Computes the Weighted Path Length of the Huffman tree, which is used to assess the optimality and correctness of student-submitted codes.

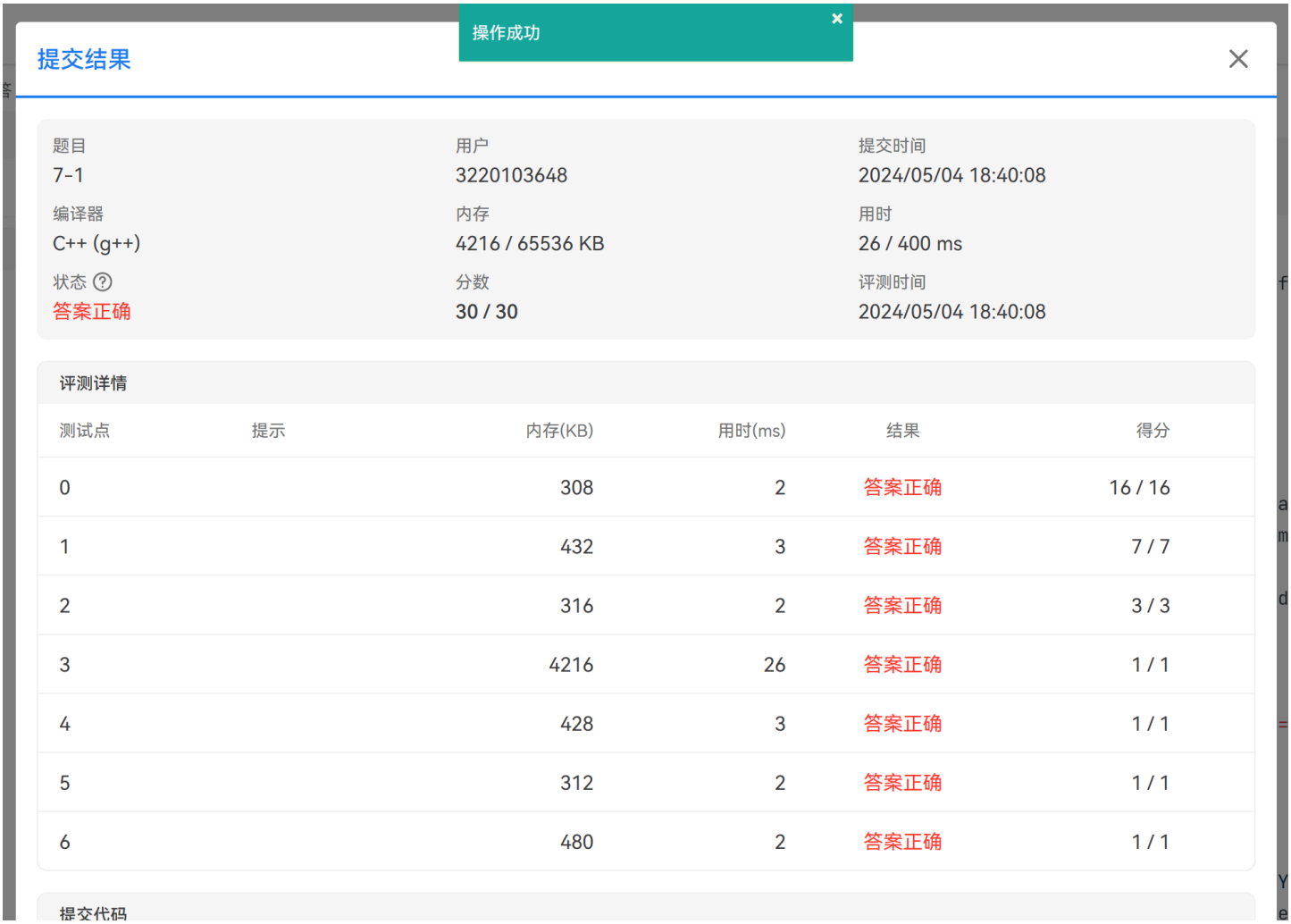
```
int WPL(struct HuffmanNode* node,int depth){
    if(node->Left == NULL && node->Right == NULL) return depth * node->weight;
    else return (WPL(node->Left,depth+1) + WPL(node->Right,depth+1)) ;
}
```

Function: judge

Validates each student's submitted encoding against the actual Huffman tree built from the input. It checks for correct encoding lengths, prefix conditions, and the total WPL.

```
bool judge(){
    struct HuffmanNode* node,* pos;
    char ch;
    string code;
    bool flag = true;  // Flag used to indicate if the submission is valid
    int j, fpos, sum_up=0; // fpos stores frequency, sum_up calculates the total WPL of the submission
    node = CreateHuffmanNode();

    for(int i = 0;i < Char_num;i++){
        cin >> ch >> code;
        pos = node; // Start from the root of the Huffman tree for each character

        // If the code length is greater than or equal to the
        // number of characters, it's automatically invalid
        if(code.length() >= Char_num) flag = false;
        else if(flag){
            for(j=0;ch!=c[j];j++);
            fpos=f[j]; // Get the frequency of the character from the array 'f'

            // Traverse the code to build or traverse down the Huffman tree
            for(j=0;j<code.length();j++){
                if(code[j]=='0'){
                    if(pos->Left==NULL) pos->Left=CreateHuffmanNode();
                    pos = pos->Left;
                }
                if(code[j]=='1'){
                    if(pos->Right==NULL) pos->Right=CreateHuffmanNode();
                    pos = pos->Right;
                }
                if(pos->weight != 0) flag = false; // If we reach a node that already has a weight,
                // the code is not a valid prefix
            }

            // After finishing the code, check if the node has children,
            // which means it's not a leaf node
            if(pos->Left || pos->Right) flag = false; // A valid leaf node shouldn't have children
            else pos->weight = fpos; // Set the weight at the leaf node

            // Calculate the contribution of this code to the total WPL
            sum_up += code.length() * pos->weight;
        }
    }

    // The calculated WPL must match the expected sum for the
    // encoding to be considered correct
    if(sum_up != sum) flag = false;
    return flag;
}
```

Chapter3: Testing Results



Chapter4: Analysis and Comments

Analyzing the time complexity and space complexity of the `judge` function involves examining each segment of the function to identify how the operations scale with input sizes. Let's break it down:

Time Complexity

1. **Initialization**:
   - Creating a new Huffman node takes constant time, $O(1)$.
2. **Reading Input and Building Huffman Tree Simulation**:
   - The loop runs `N` times, where `N` is the number of distinct characters (`Char_num`). Inside this loop:
     - Finding the index of character `ch` in array `c` involves a linear search, which takes $O(N)$ in the worst case.
     - Traversing the length of the code for each character involves operations that can be considered $O(L)$, where L is the average length of the code strings.
     - Therefore, each character processing within the loop takes $O(N + L)$.
   - Thus, the nested operations within the loop make the overall complexity for this segment $O(N \times (N + L))$.
3. **Final Check of Weighted Path Length (WPL)**:
   - Comparing `sum_up` with `sum` takes constant time, $O(1)$.

Summarizing the above, if we assume L to be less than or equal to NNN (as per typical Huffman encoding where depth generally correlates with N), the dominant term in the complexity expression becomes $O(N^2)$. This makes the overall time complexity of the `judge` function $O(N^2)$.

Space Complexity

1. **Huffman Node Allocation**:
   - A new node is allocated for each character and possibly more depending on the structure of the Huffman tree built during code traversal. The worst-case space used by the tree would be proportional to the total number of nodes created. For each character, you potentially create up to L nodes (one for each bit in the code string). Hence, space complexity can reach up to $O(N \times L)$.

2. **Auxiliary Storage**:
    - Space for storing characters, frequencies, and other variables is $O(N)$ since they are proportional to the number of distinct characters.
3. **Storage for the Huffman Tree**:
    - As previously mentioned, in the worst case where the tree becomes a skewed tree (not typical for Huffman, but possible in degenerate cases with incorrect codes), you might end up with as many nodes as the total length of all codes combined, which is $O(N \times L)$.

Therefore, the overall space complexity is $O(N \times L)$, considering the worst-case scenario where a large number of nodes are created during the validation of each student's submitted code.