LABORATORY REPORT

Algorithm Laboratory (CS-39001)

B.Tech Program in ECSc

Submitted By

Name:- SANNIDHI DEB

Roll No: 2330044

Kalinga Institute of Industrial Technology

(Deemed to be University) Bhubaneswar, India

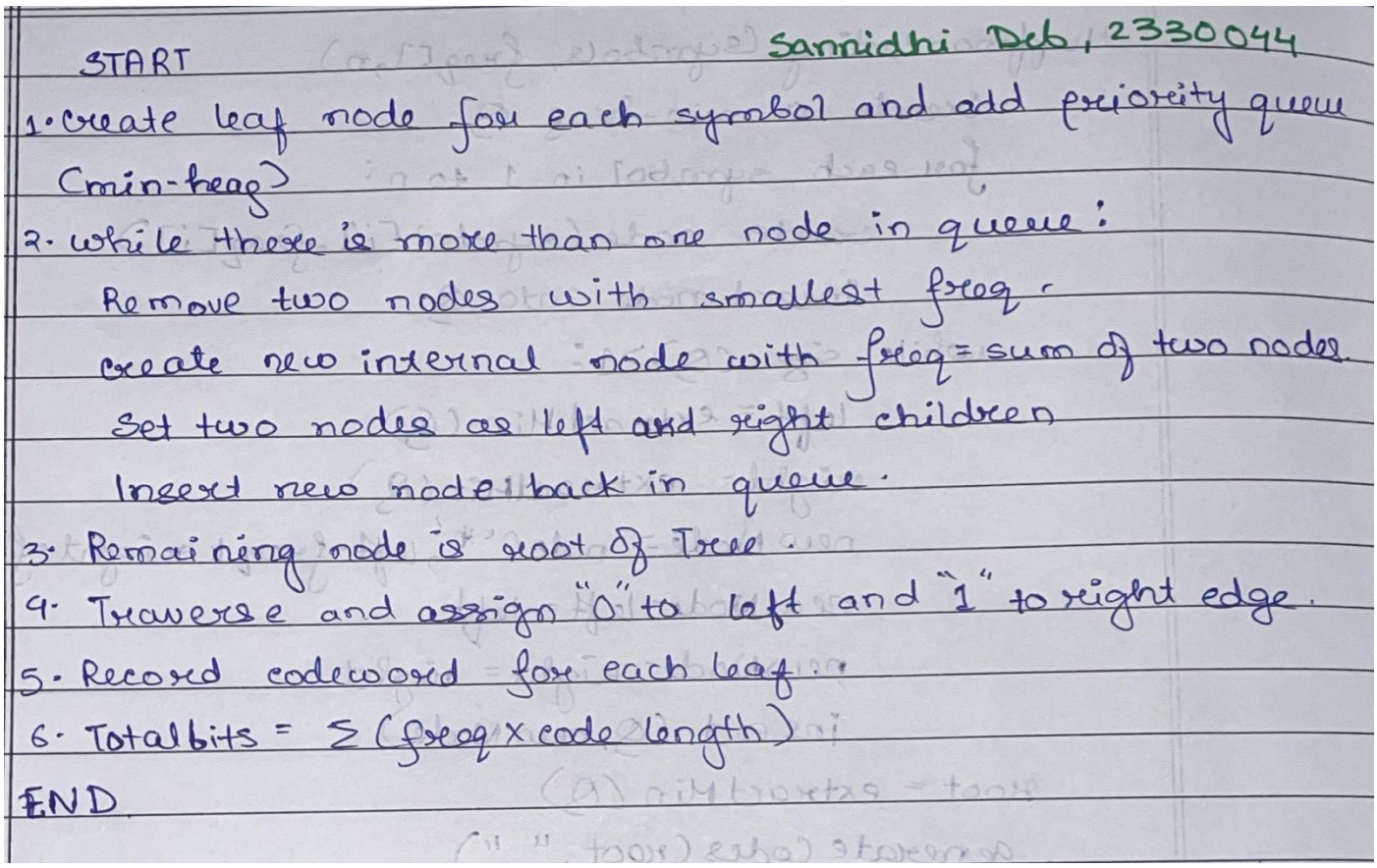
Autumn, 2025

Table of Contents

<table>
<tr><td>Experiment Number</td><td>6.1</td></tr>
<tr><td>Experiment Title</td><td>Huffman coding assigns variable length code words to fixed length input characters based on their frequencies or probabilities of occurrence. Given a set of characters along with their frequency of occurrences, write a C program to construct a Huffman tree.

Given:
<table><tr><td>Alphabet</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td></tr><tr><td>Frequency (in 1000)</td><td>45</td><td>13</td><td>12</td><td>16</td><td>9</td><td>5</td></tr></table>
Output (to be shown on the console):
<table><tr><td>Alphabet</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td></tr><tr><td>Codeword</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>BitSize</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>
Based on the frequency given, print the total number of bits needed to represent a file with 100,000 characters.</td></tr>
<tr><td>Date of Experiment</td><td>01/10/2025</td></tr>
<tr><td>Date of Submission</td><td>10/10/2025</td></tr>
</table>

1.Algorithm:-



2. Code:-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_TREE_HT 100
typedef struct MinHeapNode {
   char data;
   unsigned freq;
   struct MinHeapNode *left, *right;
} MinHeapNode;

typedef struct MinHeap {
   unsigned size;
   unsigned capacity;
   MinHeapNode** array;}
```

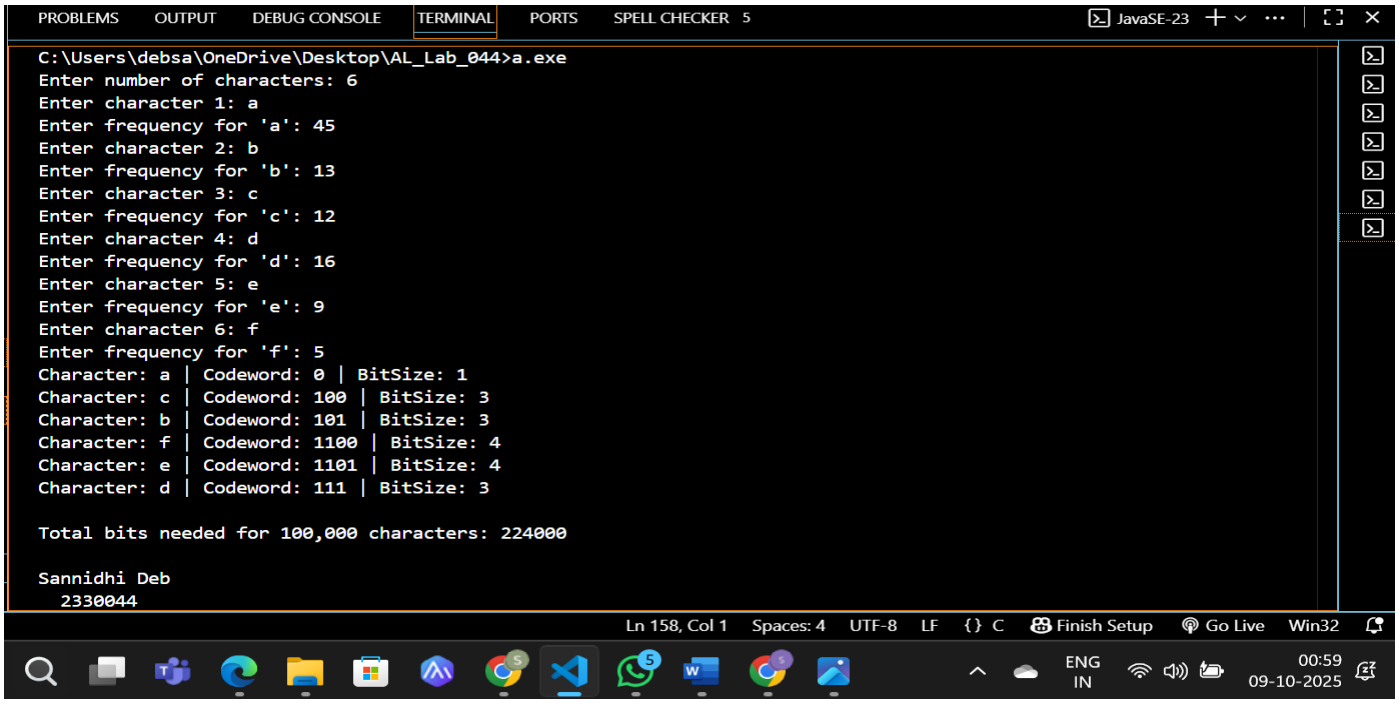
```
 MinHeap;
MinHeapNode* newNode(char data, unsigned freq) {
   MinHeapNode* temp = (MinHeapNode*)malloc(sizeof(MinHeapNode));
   temp->left = temp->right = NULL;
   temp->data = data;
   temp->freq = freq;
   return temp;
}

MinHeap* createMinHeap(unsigned capacity) {
   MinHeap* minHeap = (MinHeap*)malloc(sizeof(MinHeap));
   minHeap->size = 0;
   minHeap->capacity = capacity;
   minHeap->array = (MinHeapNode**)malloc(capacity * sizeof(MinHeapNode*));
   return minHeap;
}
void swapMinHeapNode(MinHeapNode** a, MinHeapNode** b) {
   MinHeapNode* t = *a;
   *a = *b;
   *b = t;
}
void minHeapify(MinHeap* minHeap, int idx) {
   int smallest = idx;
   int left = 2 * idx + 1;
   int right = 2 * idx + 2;
  if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
      smallest = left;

   if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
      smallest = right;

   if (smallest != idx) {
      swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
      minHeapify(minHeap, smallest);
   }
}
MinHeapNode* extractMin(MinHeap* minHeap) {
   MinHeapNode* temp = minHeap->array[0];
   minHeap->array[0] = minHeap->array[--minHeap->size];
   minHeapify(minHeap, 0);
   return temp;
}
```

```
void insertMinHeap(MinHeap* minHeap, MinHeapNode* node) {
   int i = minHeap->size++;
   while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
      minHeap->array[i] = minHeap->array[(i - 1) / 2];
      i = (i - 1) / 2;
   }
   minHeap->array[i] = node;
}

void buildMinHeap(MinHeap* minHeap) {
   for (int i = (minHeap->size - 1) / 2; i >= 0; i--)
      minHeapify(minHeap, i);
}

int isLeaf(MinHeapNode* node) {
   return !(node->left) && !(node->right);
}

MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
   MinHeap* minHeap = createMinHeap(size);
   for (int i = 0; i < size; i++)
      minHeap->array[i] = newNode(data[i], freq[i]);
   minHeap->size = size;
   buildMinHeap(minHeap);
   return minHeap;
}

MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
   MinHeapNode *left, *right, *top;
   MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

   while (minHeap->size > 1) {
      left = extractMin(minHeap);
      right = extractMin(minHeap);

      top = newNode('$', left->freq + right->freq);
      top->left = left;
      top->right = right;

      insertMinHeap(minHeap, top);
   }
  return extractMin(minHeap);
```

```
}

void printCodes(MinHeapNode* root, int arr[], int top, int* totalBits) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1, totalBits);
    }
if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1, totalBits);
    }

    if (isLeaf(root)) {
        printf("Character: %c\n", root->data);
        printf("Codeword: ");
        for (int i = 0; i < top; i++)
            printf("%d", arr[i]);
        printf("\nBitSize: %d\n\n", top);
        *totalBits += root->freq * top;
    }
}

void HuffmanCodes(char data[], int freq[], int size) {
    MinHeapNode* root = buildHuffmanTree(data, freq, size);
    int arr[MAX_TREE_HT], top = 0, totalBits = 0;
    printCodes(root, arr, top, &totalBits);
    printf("Total bits needed for 100,000 characters: %d\n", totalBits);
}
int main() {
    int n;
    printf("Enter number of characters:\n");
    scanf("%d", &n);
  char data[n];
    int freq[n];
for (int i = 0; i < n; i++) {
        printf("Enter character %d:\n", i + 1);
        scanf(" %c", &data[i]);
        printf("Enter frequency for '%c':\n", data[i]);
        scanf("%d", &freq[i]);
    }
 HuffmanCodes(data, freq, n);
```

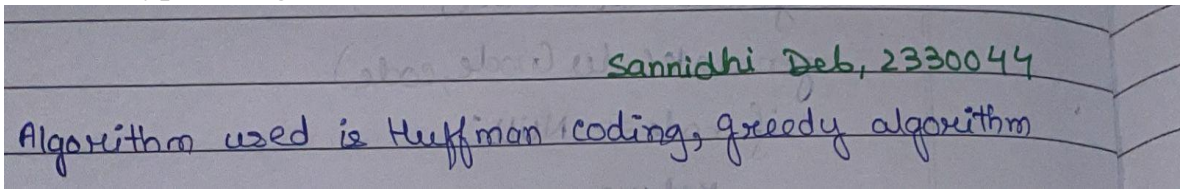
```
    printf("\nSannidhi Deb\nRoll No: 2330044\n");
    return 0;
}
```

3.Results/Output:- Entire Screen Shot including Date & Time:-

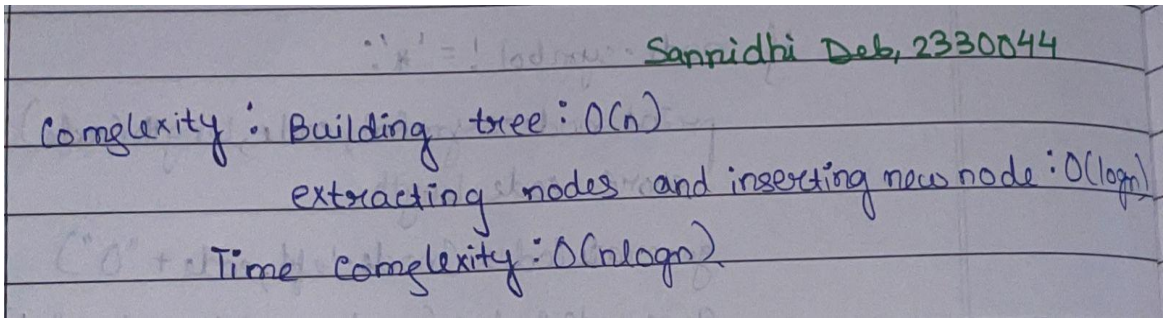


4. Remarks:-

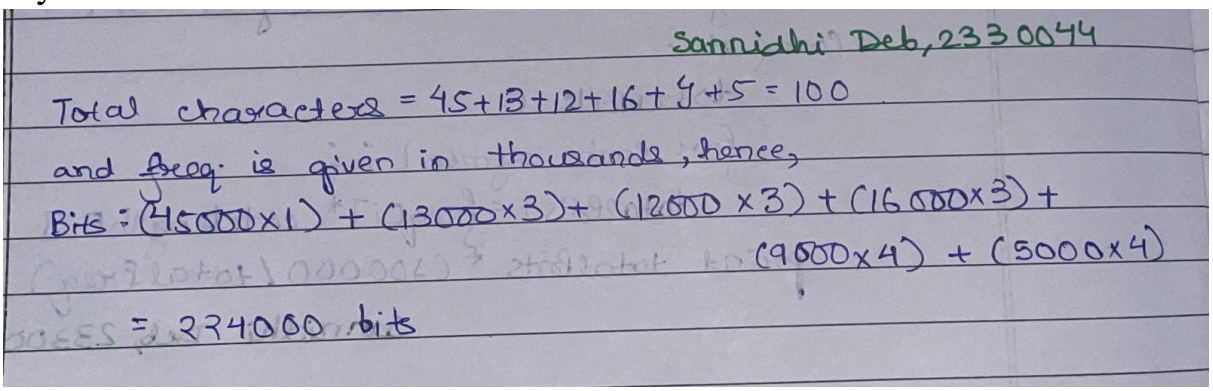
1. What type of algorithm is used?



2. Analyze the complexity of your algorithm.

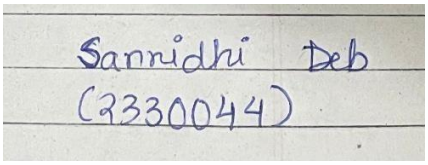


3.Any other observations?



5. Conclusion:-

Huffman coding efficiently compresses data by assigning shorter codes to more frequent characters and longer codes to less frequent ones. It minimizes the total number of bits required for encoding, making it a powerful application of greedy algorithms in lossless data compression.



Signature of the FIC

Sannidhi Deb

(Name of the FIC)