Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Name: Apurba Koirala

Reg no: 22BCE3799

Subject Code: BCSE204P

Course Title: Design and Analysis of Algorithms Lab

Lab Slot: L39 + L40

Guided by: Dr. IYAPPAN P

Lab Assessment 1

1. **Implement Fractional Knapsack Problem using Greedy strategy for the following instance:**

   n = 5

   w = 60 kg

   (w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)

   (b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)

Your program should consist of the Following:

Total Number of weight

Size of knapsack

Getting the Weight to be inserted in Knapsack

Getting the Profit of weight

Displaying the weight and its profit before sorting

Calculating the ratio of Weight and Profit

Sorting the Ratio in descending order

Displaying the weight and its profit after sorting

Print the Optimal Solution with how much of weights taken

Algorithm:
Input profits (P[]), weights (W[]), and max capacity (M).
Calculate profit-to-weight ratios (X[]).
Sort X[], P[], and W[] in descending order of X[].
Initialize rem_cap = M and profit = 0.
Iteratively take full or fractional items until rem_cap = 0.
Calculate total profit as profit += R[i] * P[i].
Output the total profit.

Source Code:

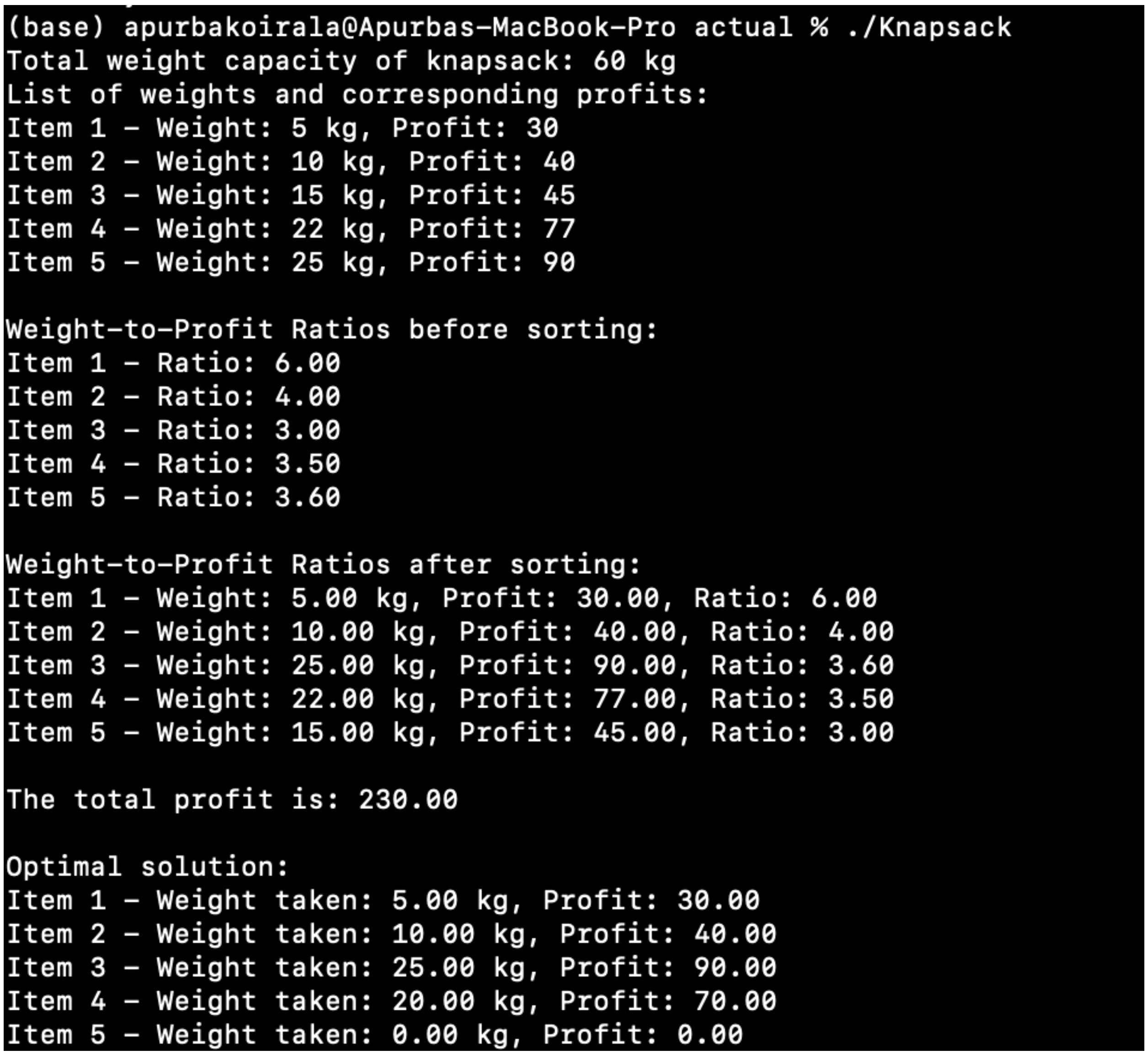
```
#include <iostream>
#include <iomanip>
using namespace std;

double FractionalKnapsack(double P[], double W[], double X[], double R[], int M, int n) {
    int rem_cap = M;
    for (int i = 0; i < n; i++) {
        R[i] = 0;
    }

    double profit = 0;
    for (int i = 0; i < n; i++) {
        if (W[i] <= rem_cap) {
            R[i] = 1;
            rem_cap = rem_cap - W[i];
        } else {
            R[i] = double(rem_cap) / W[i];
            rem_cap = 0;
        }
    }

    for (int i = 0; i < n; i++) {
        profit += R[i] * P[i];
    }
    return profit;
}

void DescSort(double arr[], double P[], double W[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] < arr[j + 1]) {
                double temp = arr[j + 1];
                double val1 = P[j + 1];
                double val2 = W[j + 1];

                arr[j + 1] = arr[j];
                P[j + 1] = P[j];
                W[j + 1] = W[j];

                arr[j] = temp;
                P[j] = val1;
                W[j] = val2;
            }
        }
    }
}

int main() {
    int n = 5;
    int max = 60;
    double P[] = {30, 40, 45, 77, 90};
    double W[] = {5, 10, 15, 22, 25};
    double R[n];
    double X[n];

    cout << "Total weight capacity of knapsack: " << max << " kg" << endl;
    cout << "List of weights and corresponding profits:" << endl;

    for (int i = 0; i < n; i++) {
        cout << "Item " << i + 1 << " - Weight: " << W[i] << " kg, Profit: " << P[i] << endl;
    }

    for (int i = 0; i < n; i++) {
        X[i] = (P[i] / W[i]);
    }

    cout << "\nWeight-to-Profit Ratios before sorting:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Item " << i + 1 << " - Ratio: " << fixed << setprecision(2) << X[i] << endl;
    }

    DescSort(X, P, W, n);

    cout << "\nWeight-to-Profit Ratios after sorting:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Item " << i + 1 << " - Weight: " << W[i] << " kg, Profit: " << P[i] << ", Ratio: " << fixed << setprecision(2)
<< X[i] << endl;
    }
```

```cpp
    double profit = FractionalKnapsack(P, W, X, R, max, n);
    cout << "\nThe total profit is: " << fixed << setprecision(2) << profit << endl;

    cout << "\nOptimal solution:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Item " << i + 1 << " - Weight taken: " << R[i] * W[i] << " kg, Profit: " << R[i] * P[i] << endl;
    }

    return 0;
}
```

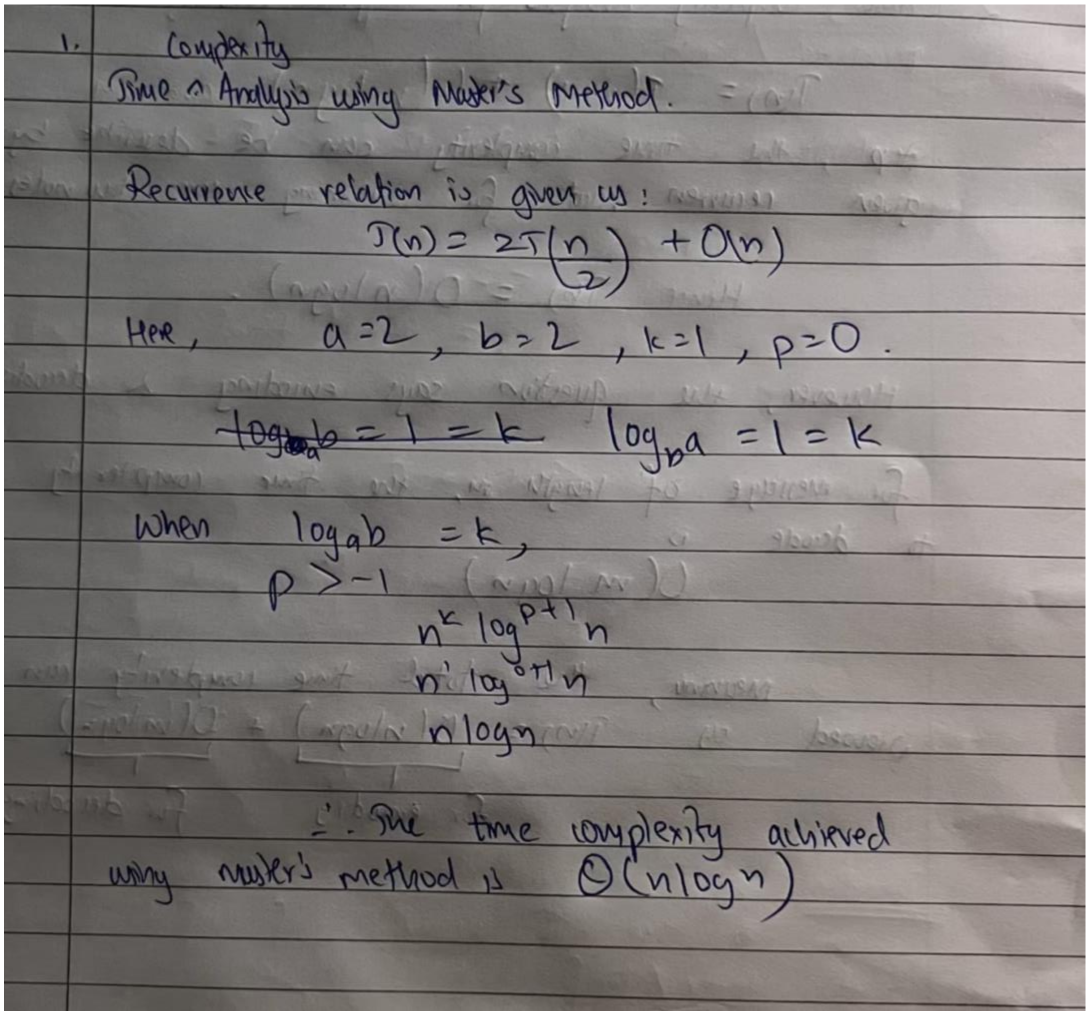
Input:

```cpp
double P[] = {30, 40, 45, 77, 90};
double W[] = {5, 10, 15, 22, 25};
```

Output:


```
(base) apurbakoirala@Apurbas-MacBook-Pro actual % ./Knapsack
Total weight capacity of knapsack: 60 kg
List of weights and corresponding profits:
Item 1 - Weight: 5 kg, Profit: 30
Item 2 - Weight: 10 kg, Profit: 40
Item 3 - Weight: 15 kg, Profit: 45
Item 4 - Weight: 22 kg, Profit: 77
Item 5 - Weight: 25 kg, Profit: 90

Weight-to-Profit Ratios before sorting:
Item 1 - Ratio: 6.00
Item 2 - Ratio: 4.00
Item 3 - Ratio: 3.00
Item 4 - Ratio: 3.50
Item 5 - Ratio: 3.60

Weight-to-Profit Ratios after sorting:
Item 1 - Weight: 5.00 kg, Profit: 30.00, Ratio: 6.00
Item 2 - Weight: 10.00 kg, Profit: 40.00, Ratio: 4.00
Item 3 - Weight: 25.00 kg, Profit: 90.00, Ratio: 3.60
Item 4 - Weight: 22.00 kg, Profit: 77.00, Ratio: 3.50
Item 5 - Weight: 15.00 kg, Profit: 45.00, Ratio: 3.00

The total profit is: 230.00

Optimal solution:
Item 1 - Weight taken: 5.00 kg, Profit: 30.00
Item 2 - Weight taken: 10.00 kg, Profit: 40.00
Item 3 - Weight taken: 25.00 kg, Profit: 90.00
Item 4 - Weight taken: 20.00 kg, Profit: 70.00
Item 5 - Weight taken: 0.00 kg, Profit: 0.00
```

Time Complexity:



1. Time complexity Analysis using Master's Method.

Recurrence relation is given as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Here, $a=2$, $b=2$, $k=1$, $p=0$.

~~$\log_a b = 1 = k$~~ $\log_b a = 1 = k$

When $\log_a b = k$,

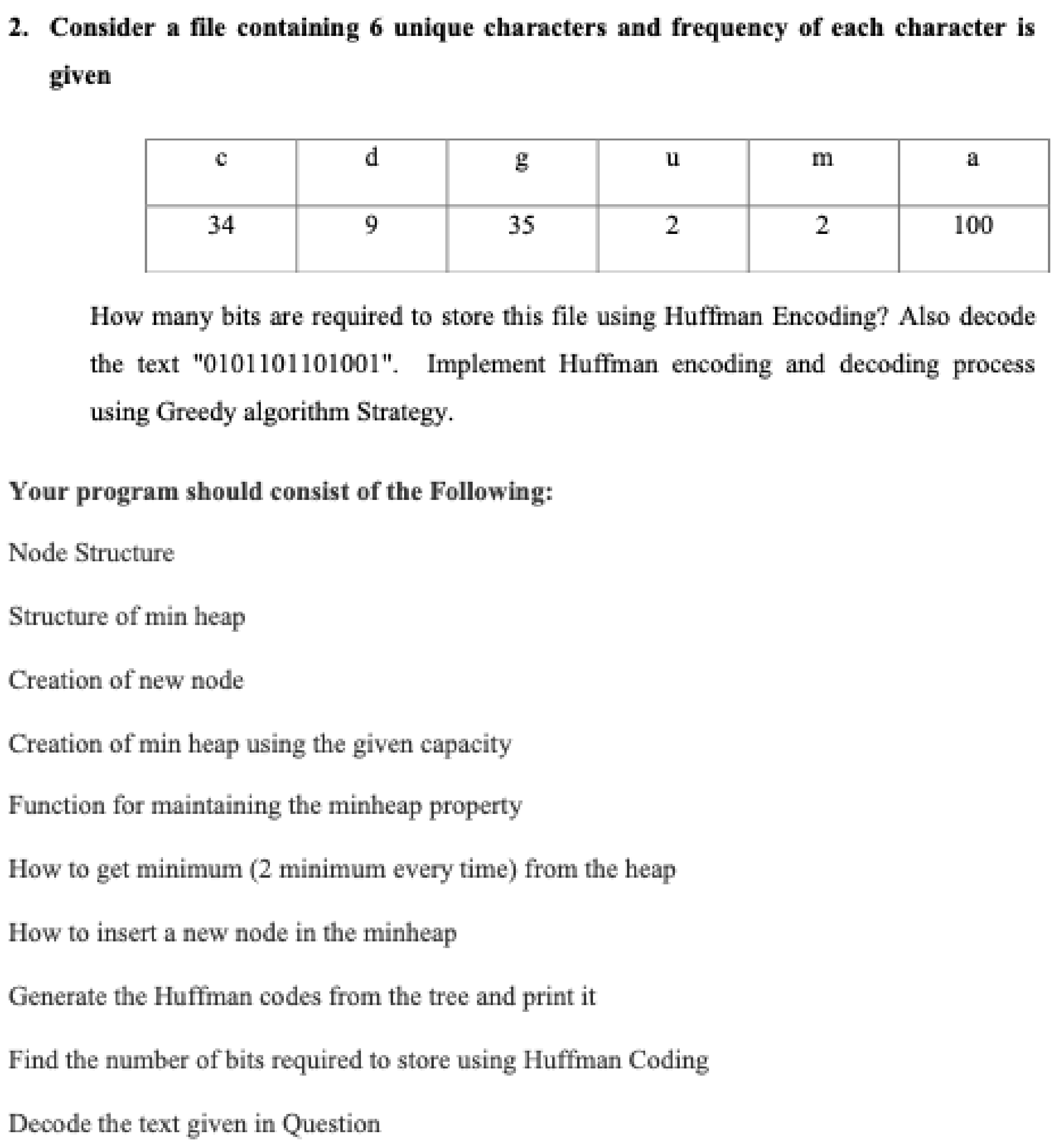
$p > -1$

$n^k \log^{p+1} n$

$n^1 \log^{0+1} n$

$n \log n$

$\therefore$ The time complexity achieved using master's method is $\Theta(n \log n)$



2. **Consider a file containing 6 unique characters and frequency of each character is given**

c	d	g	u	m	a
34	9	35	2	2	100

How many bits are required to store this file using Huffman Encoding? Also decode the text "0101101101001". Implement Huffman encoding and decoding process using Greedy algorithm Strategy.

Your program should consist of the Following:

Node Structure

Structure of min heap

Creation of new node

Creation of min heap using the given capacity

Function for maintaining the minheap property

How to get minimum (2 minimum every time) from the heap

How to insert a new node in the minheap

Generate the Huffman codes from the tree and print it

Find the number of bits required to store using Huffman Coding

Decode the text given in Question

Algorithm:

Initialize variables and input item weights and profits.

Calculate the weight-to-profit ratio for each item.

Display the weights and profits before sorting.

Sort the items based on the weight-to-profit ratio in descending order.

Display the sorted weights, profits, and ratios.

Implement the Fractional Knapsack algorithm to calculate the total profit, taking items greedily based on the sorted ratios.

Display the total profit.

Display the optimal solution with the amount of each item taken and its corresponding profit.

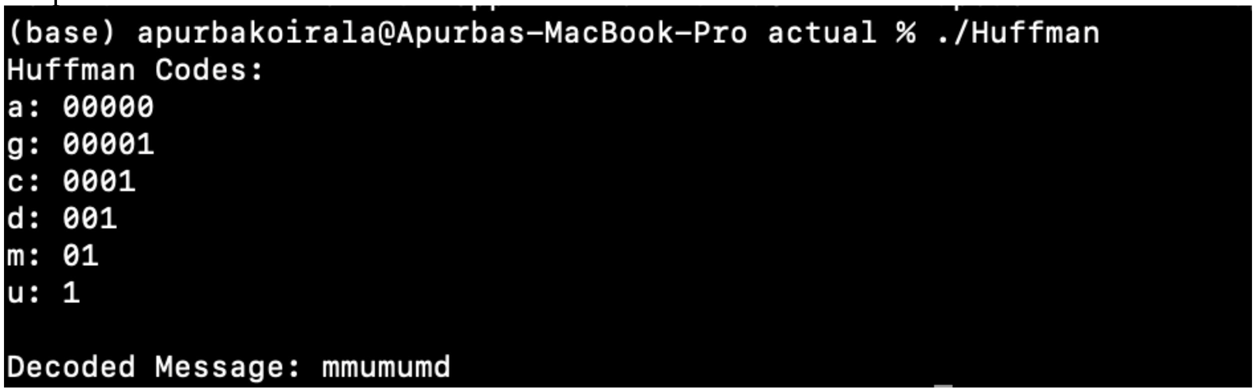
Source Code:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct HuffmanNode {
    char data;
    int freq;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(char d, int f) : data(d), freq(f), left(nullptr), right(nullptr) {}
};

vector<HuffmanNode*> createLeaves(char data[], int frequency[], int size) {
    vector<HuffmanNode*> nodes;
    for (int i = 0; i < size; i++) {
        nodes.push_back(new HuffmanNode(data[i], frequency[i]));
    }
    return nodes;
}

void swap(HuffmanNode*& a, HuffmanNode*& b) {
    HuffmanNode* temp = a;
    a = b;
    b = temp;
}

void heapify(vector<HuffmanNode*>& arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l]->freq > arr[largest]->freq)
        largest = l;
    if (r < n && arr[r]->freq > arr[largest]->freq)
        largest = r;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<HuffmanNode*>& arr, int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

HuffmanNode* buildTree(vector<HuffmanNode*>& nodes) {
    int size = nodes.size();
    while (size > 1) {
        heapSort(nodes, size);

        HuffmanNode* left = nodes[size - 1];
        HuffmanNode* right = nodes[size - 2];
        HuffmanNode* combined = new HuffmanNode('=', left->freq + right->freq);

        combined->left = left;
        combined->right = right;

        nodes[size - 2] = combined;
        size--;
    }
    return nodes[0];
}

void encode(HuffmanNode* root, string code) {
    if (!root) return;
    if (root->data != '=')
        cout << root->data << ": " << code << endl;
    encode(root->left, code + "0");
    encode(root->right, code + "1");
}

char decode(HuffmanNode* root, string& encoded, int& index) {
    if (!root->left && !root->right)
        return root->data;
    if (encoded[index] == '0') {
        index++;
        return decode(root->left, encoded, index);
    } else {
        index++;
        return decode(root->right, encoded, index);
    }
}

string decodeString(HuffmanNode* root, string& encoded) {
    string result = "";
    int index = 0;
    while (index < encoded.length()) {
        result += decode(root, encoded, index);
    }
    return result;
}

int main() {
    char chars[] = {'c', 'd', 'g', 'u', 'm', 'a'};
    int frequencies[] = {34, 9, 35, 2, 2, 100};
    int size = sizeof(frequencies) / sizeof(int);
    vector<HuffmanNode*> leaves = createLeaves(chars, frequencies, size);
    HuffmanNode* root = buildTree(leaves);

    cout << "Huffman Codes:" << endl;
    encode(root, "");

    string encodedMessage = "0101101101001";
    string decodedMessage = decodeString(root, encodedMessage);

    cout << "\nDecoded Message: " << decodedMessage << endl;
    return 0;
}
```

Input:

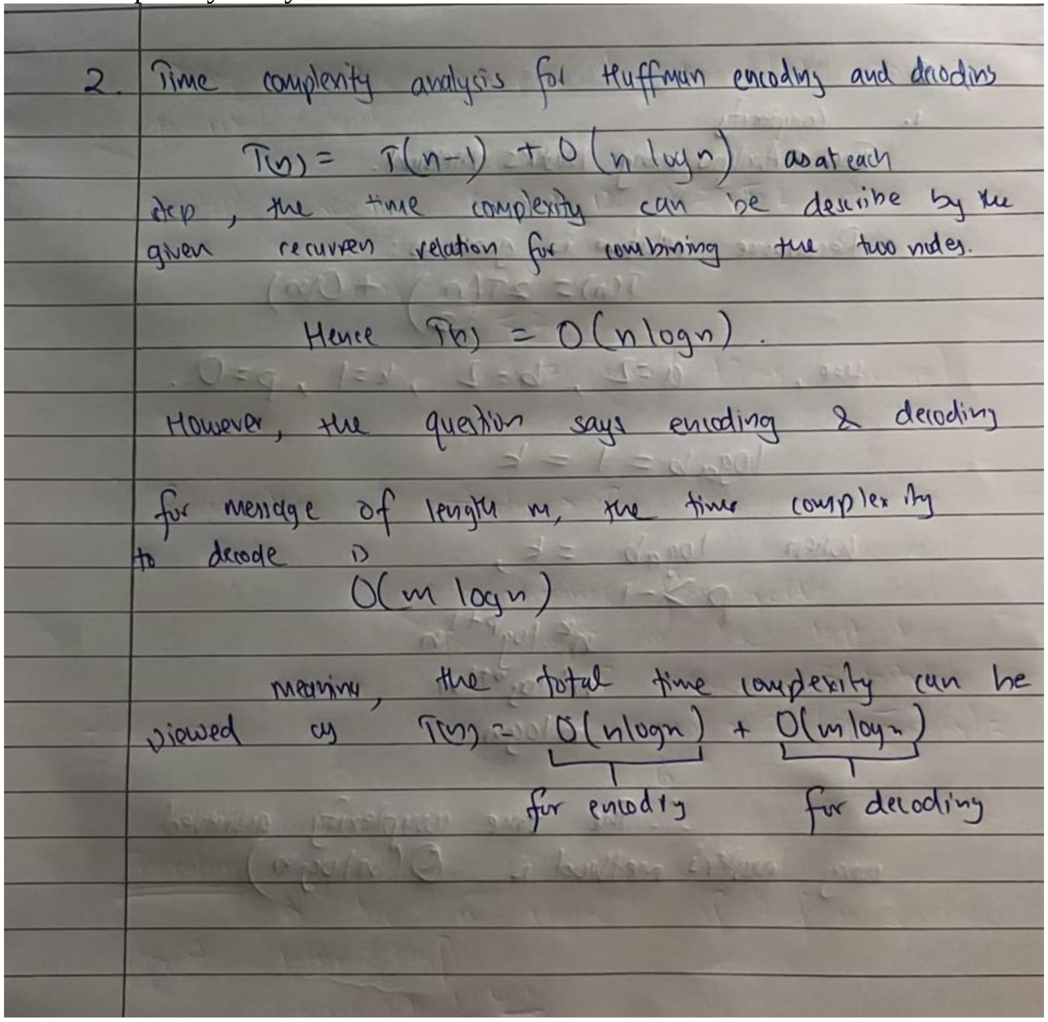
```
    char chars[] = {'c', 'd', 'g', 'u', 'm', 'a'};
    int frequencies[] = {34, 9, 35, 2, 2, 100};
    int size = sizeof(frequencies) / sizeof(int);
```

Output:



```
(base) apurbakoirala@Apurbas-MacBook-Pro actual % ./Huffman
Huffman Codes:
a: 00000
g: 00001
c: 0001
d: 001
m: 01
u: 1

Decoded Message: mmumumd
```

Time complexity analysis:



2. Time complexity analysis for Huffman encoding and decoding

$T(n) = T(n-1) + O(n \log n)$ as at each step, the time complexity can be describe by the given recurren relation for combining the two nodes.

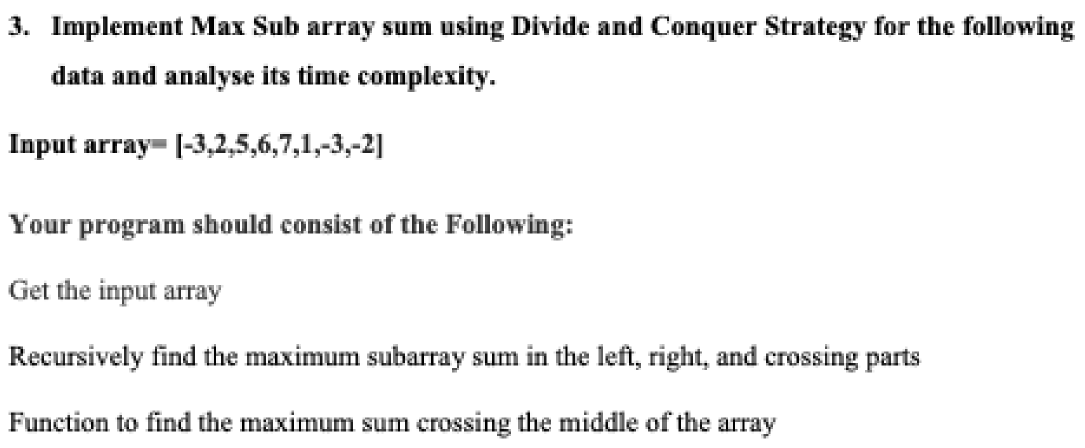
Hence $T(n) = O(n \log n)$.

However, the question says encoding & decoding

for message of length m, the time complexity to decode is

$O(m \log n)$

meaning, the total time complexity can be viewed as $T(n) = \underbrace{O(n \log n)}_{\text{for encoding}} + \underbrace{O(m \log n)}_{\text{for decoding}}$


3. **Implement Max Sub array sum using Divide and Conquer Strategy for the following data and analyse its time complexity.**

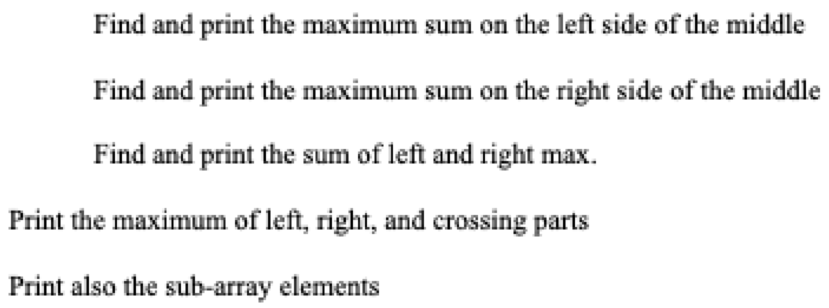
Input array= [-3,2,5,6,7,1,-3,-2]

Your program should consist of the Following:

Get the input array

Recursively find the maximum subarray sum in the left, right, and crossing parts

Function to find the maximum sum crossing the middle of the array


Find and print the maximum sum on the left side of the middle

Find and print the maximum sum on the right side of the middle

Find and print the sum of left and right max.

Print the maximum of left, right, and crossing parts

Print also the sub-array elements

Algorithm:
Divide the array into two halves at the middle index.
Recursively find the maximum subarray sum in the left half.
Recursively find the maximum subarray sum in the right half.
Find the maximum sum subarray that crosses the middle.
Compare the maximum sums from the left, right, and crossing subarrays.
Return the maximum sum along with the indices of the subarray.
If the subarray has one element, return its value as the sum and its index.

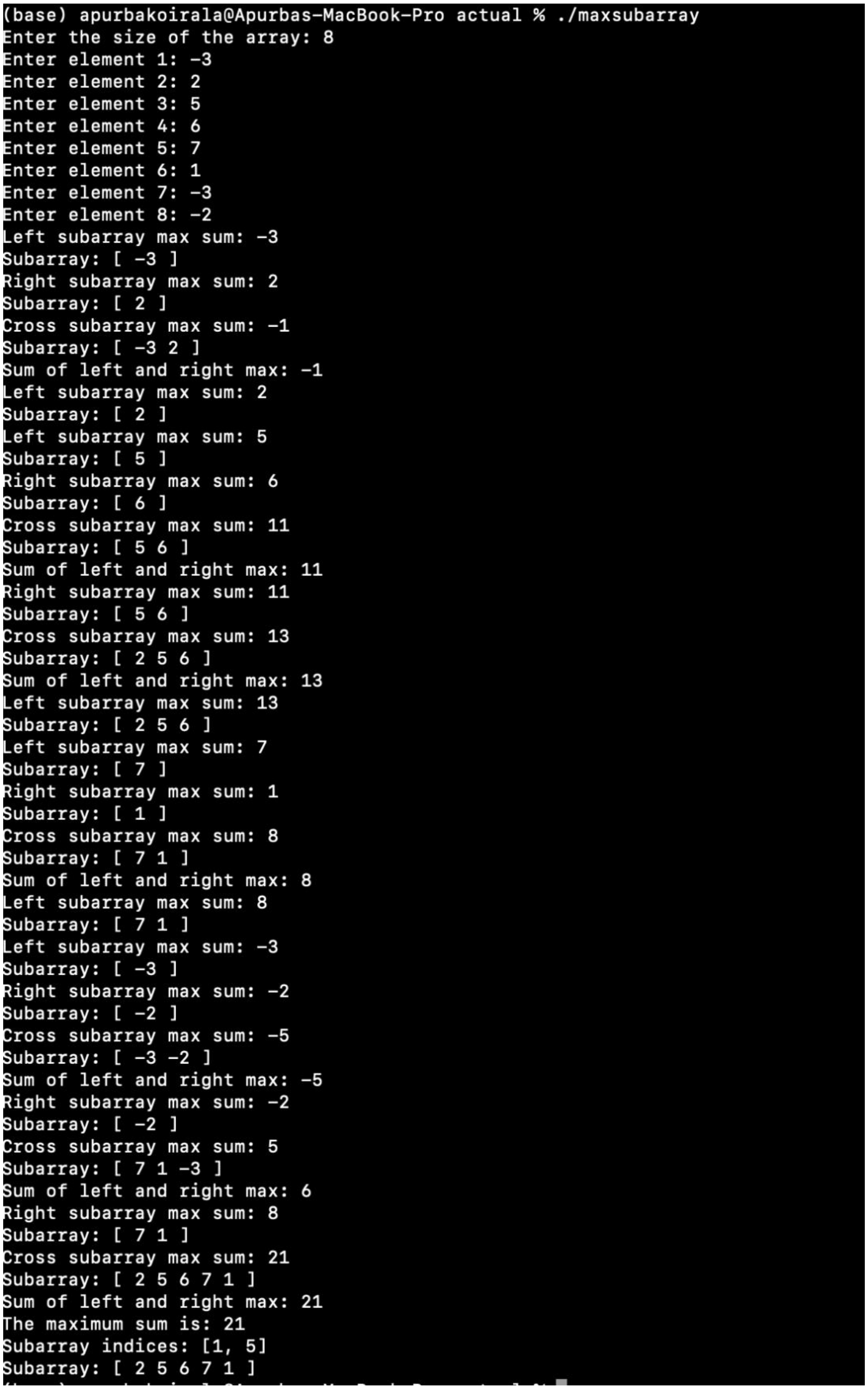
Source Code:

```
#include <iostream>
#include <limits>
using namespace std;
```

```cpp
void findMaxSubarray(int A[], int low, int high, int& resultLow, int& resultHigh, int& resultSum);
void findMaxCrossSubarray(int A[], int low, int mid, int high, int& resultLow, int& resultHigh, int& resultSum);

void printSubarray(int A[], int low, int high) {
    cout << "Subarray: [ ";
    for (int i = low; i <= high; i++) {
        cout << A[i] << " ";
    }
    cout << "]" << endl;
}

void findMaxSubarray(int A[], int low, int high, int& resultLow, int& resultHigh, int& resultSum) {
    if (high == low) {
        resultLow = low;
        resultHigh = high;
        resultSum = A[low];
    } else {
        int mid = (low + high) / 2;

        int leftLow, leftHigh, leftSum;
        findMaxSubarray(A, low, mid, leftLow, leftHigh, leftSum);
        cout << "Left subarray max sum: " << leftSum << endl;
        printSubarray(A, leftLow, leftHigh);

        int rightLow, rightHigh, rightSum;
        findMaxSubarray(A, mid + 1, high, rightLow, rightHigh, rightSum);
        cout << "Right subarray max sum: " << rightSum << endl;
        printSubarray(A, rightLow, rightHigh);

        int crossLow, crossHigh, crossSum;
        findMaxCrossSubarray(A, low, mid, high, crossLow, crossHigh, crossSum);
        cout << "Cross subarray max sum: " << crossSum << endl;
        printSubarray(A, crossLow, crossHigh);

        cout << "Sum of left and right max: " << leftSum + rightSum << endl;

        if (leftSum >= rightSum && leftSum >= crossSum) {
            resultLow = leftLow;
            resultHigh = leftHigh;
            resultSum = leftSum;
        } else if (rightSum >= leftSum && rightSum >= crossSum) {
            resultLow = rightLow;
            resultHigh = rightHigh;
            resultSum = rightSum;
        } else {
            resultLow = crossLow;
            resultHigh = crossHigh;
            resultSum = crossSum;
        }
    }
}

void findMaxCrossSubarray(int A[], int low, int mid, int high, int& resultLow, int& resultHigh, int& resultSum) {
    int leftSum = numeric_limits<int>::min();
    int sum = 0;
    int maxLeft;

    for (int i = mid; i >= low; i--) {
        sum += A[i];
        if (sum > leftSum) {
            leftSum = sum;
            maxLeft = i;
        }
    }

    int rightSum = numeric_limits<int>::min();
    sum = 0;
    int maxRight;

    for (int j = mid + 1; j <= high; j++) {
        sum += A[j];
        if (sum > rightSum) {
            rightSum = sum;
            maxRight = j;
        }
    }

    resultLow = maxLeft;
    resultHigh = maxRight;
    resultSum = leftSum + rightSum;
}

int main() {
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;

    int* arr = new int[size];

    for (int i = 0; i < size; i++) {
        cout << "Enter element " << i + 1 << ": ";
        cin >> arr[i];
    }

    int resultLow, resultHigh, resultSum;
    findMaxSubarray(arr, 0, size - 1, resultLow, resultHigh, resultSum);

    cout << "The maximum sum is: " << resultSum << endl;
    cout << "Subarray indices: [" << resultLow << ", " << resultHigh << "]" << endl;
    printSubarray(arr, resultLow, resultHigh);

    delete[] arr;
    return 0;
}
```

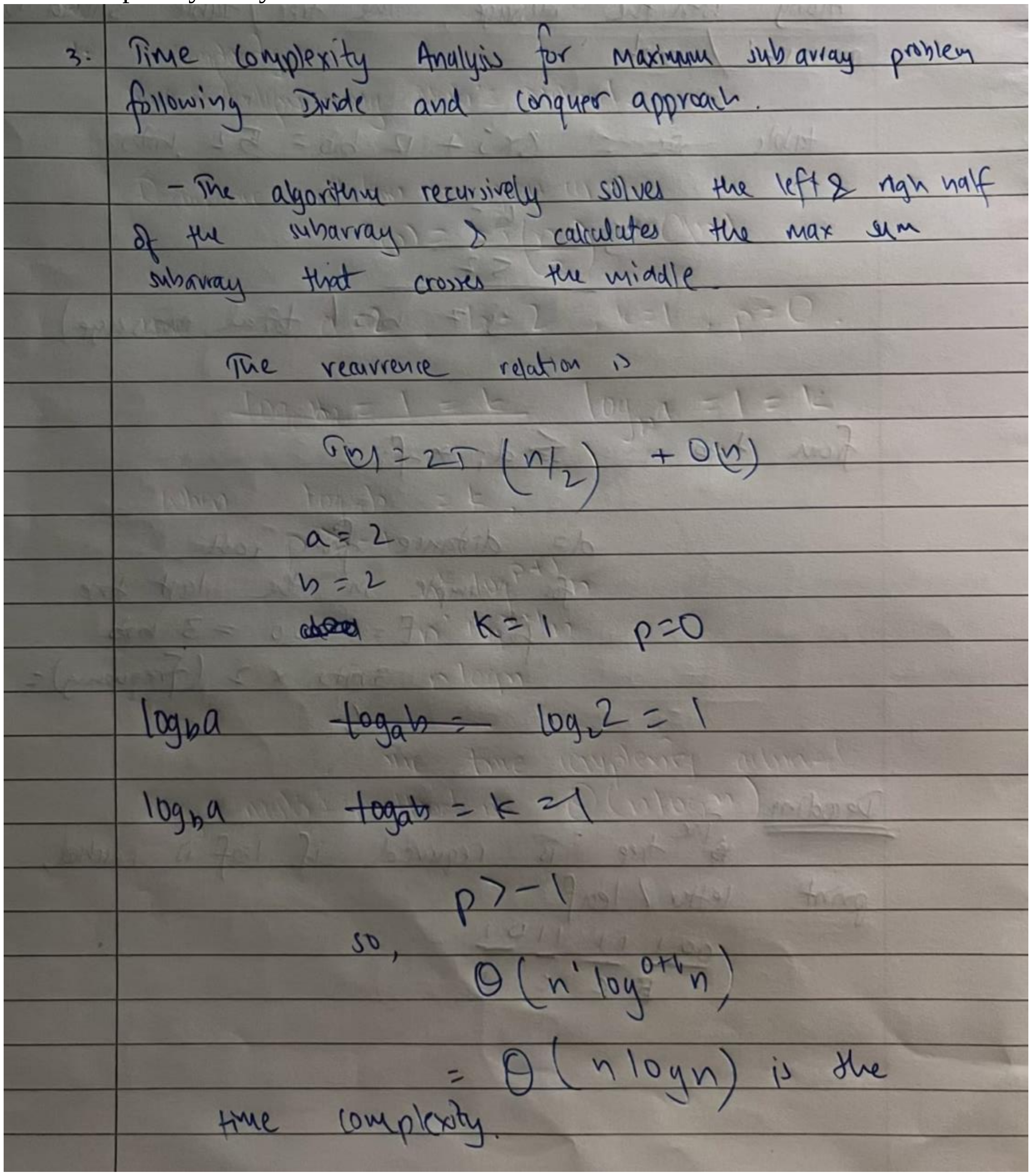
Input:
Size of array = 8
Array = [-3, 2, 5, 6, 7, 1, -3, -2]

Output:



```
(base) apurbakoirala@Apurbas-MacBook-Pro actual % ./maxsubarray
Enter the size of the array: 8
Enter element 1: -3
Enter element 2: 2
Enter element 3: 5
Enter element 4: 6
Enter element 5: 7
Enter element 6: 1
Enter element 7: -3
Enter element 8: -2
Left subarray max sum: -3
Subarray: [ -3 ]
Right subarray max sum: 2
Subarray: [ 2 ]
Cross subarray max sum: -1
Subarray: [ -3 2 ]
Sum of left and right max: -1
Left subarray max sum: 2
Subarray: [ 2 ]
Left subarray max sum: 5
Subarray: [ 5 ]
Right subarray max sum: 6
Subarray: [ 6 ]
Cross subarray max sum: 11
Subarray: [ 5 6 ]
Sum of left and right max: 11
Right subarray max sum: 11
Subarray: [ 5 6 ]
Cross subarray max sum: 13
Subarray: [ 2 5 6 ]
Sum of left and right max: 13
Left subarray max sum: 13
Subarray: [ 2 5 6 ]
Left subarray max sum: 7
Subarray: [ 7 ]
Right subarray max sum: 1
Subarray: [ 1 ]
Cross subarray max sum: 8
Subarray: [ 7 1 ]
Sum of left and right max: 8
Left subarray max sum: 8
Subarray: [ 7 1 ]
Left subarray max sum: -3
Subarray: [ -3 ]
Right subarray max sum: -2
Subarray: [ -2 ]
Cross subarray max sum: -5
Subarray: [ -3 -2 ]
Sum of left and right max: -5
Right subarray max sum: -2
Subarray: [ -2 ]
Cross subarray max sum: 5
Subarray: [ 7 1 -3 ]
Sum of left and right max: 6
Right subarray max sum: 8
Subarray: [ 7 1 ]
Cross subarray max sum: 21
Subarray: [ 2 5 6 7 1 ]
Sum of left and right max: 21
The maximum sum is: 21
Subarray indices: [1, 5]
Subarray: [ 2 5 6 7 1 ]
```

Time complexity analysis:



3: Time complexity Analysis for Maximum sub array problem following Divide and conquer approach.

– The algorithm recursively solves the left & righ half of the subarray & calculates the max sum subarray that crosses the middle.

The recurrence relation is

$$T(n) = 2T\left(n/_2\right) + O(n)$$

$a = 2$

$b = 2$

~~[illegible]~~ $k = 1$ $p = 0$

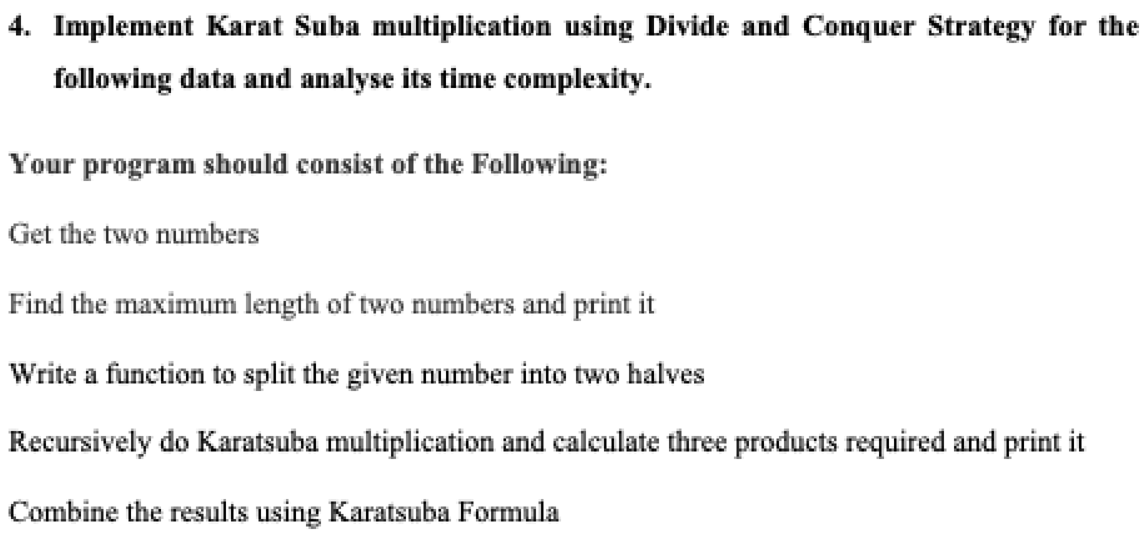
$\log_b a$ ~~$\log_a b =$~~ $\log_2 2 = 1$

$\log_b a$ ~~$\log_a b$~~ $= k = 1$

$p > -1$

so, $\Theta(n^1 \log^{0+1} n)$

$= \Theta(n \log n)$ is the time complexity.


4. Implement Karat Suba multiplication using Divide and Conquer Strategy for the following data and analyse its time complexity.

Your program should consist of the Following:

Get the two numbers

Find the maximum length of two numbers and print it

Write a function to split the given number into two halves

Recursively do Karatsuba multiplication and calculate three products required and print it

Combine the results using Karatsuba Formula

Algorithm:

Take two numbers as input from the user.

Find the maximum length of the two numbers by comparing their digit lengths.

Split each number into two halves: the higher half (most significant digits) and the lower half (least significant digits).

If the numbers are small enough (single-digit), directly multiply them and return the result (base case).

For larger numbers, recursively calculate three products:

Multiply the lower halves of both numbers (low1 × low2).

Multiply the sum of the lower and higher halves of both numbers (low1 + high1) × (low2 + high2).

Multiply the higher halves of both numbers (high1 × high2).

Combine the three results using the Karatsuba formula:

Result = (high1 × high2) * 10^(2 * half-length) + ((low1 + high1) × (low2 + high2) - high1 × high2 - low1 × low2) * 10^half-length + (low1 × low2).

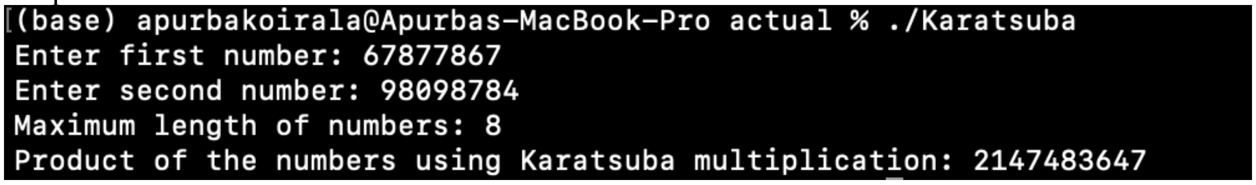
Return the final result as the product of the two numbers.

Source Code:

```cpp
#include <iostream>
#include <cmath>
#include <string>
#include <algorithm>

using namespace std;

int maxLength(int a, int b) {
    return max(to_string(a).length(), to_string(b).length());
}

pair<int, int> splitNumber(int num, int n) {
    int power = pow(10, n / 2);
    int low = num % power;
    int high = num / power;
    return make_pair(high, low);
}

int karatsuba(int x, int y) {
    int length = maxLength(x, y);

    if (length == 1) {
        return x * y;
    }

    int n = length;
    int half = n / 2;

    pair<int, int> x_split = splitNumber(x, n);
    pair<int, int> y_split = splitNumber(y, n);

    int high1 = x_split.first, low1 = x_split.second;
    int high2 = y_split.first, low2 = y_split.second;
    int z0 = karatsuba(low1, low2);
    int z1 = karatsuba(low1 + high1, low2 + high2);
    int z2 = karatsuba(high1, high2);

    return (z2 * pow(10, 2 * half)) + ((z1 - z2 - z0) * pow(10, half)) + z0;
}

int main() {
    int x, y;

    cout << "Enter first number: ";
    cin >> x;
    cout << "Enter second number: ";
    cin >> y;

    int max_len = maxLength(x, y);
    cout << "Maximum length of numbers: " << max_len << endl;

    int result = karatsuba(x, y);
    cout << "Product of the numbers using Karatsuba multiplication: " << result << endl;

    return 0;
}
```

Input:
First number: 67877867

Second Number: 98098784

Output:


```
[(base) apurbakoirala@Apurbas-MacBook-Pro actual % ./Karatsuba
 Enter first number: 67877867
 Enter second number: 98098784
 Maximum length of numbers: 8
 Product of the numbers using Karatsuba multiplication: 2147483647
```

Time complexity analysis:

4. Time complexity analysis for Karatsuba using Master's Method.

The algorithm divides the problem into three sub-problems, each with ~~the~~ number of size $n/2$.

Size of subproblem $= n/2$.
3 subproblem.

Combining takes $O(n)$ time.

Here,
$$3T\left(\frac{n}{2}\right) + O(n)$$

here,
$$a = 3, \quad b = 2 \quad p = 0, \; k = 1$$

$$\log_b a = \log_2 3 = 1.585$$

$$\log_b a > k = 1$$

hence, $O\left(n^{\log_b a}\right)$

$= O\left(n^{1.585}\right)$ is the time complexity of the algorithm.