**A blue and white logo

Description automatically generated**

# 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

A screenshot of a white paper

Description automatically generated

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;

}

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;

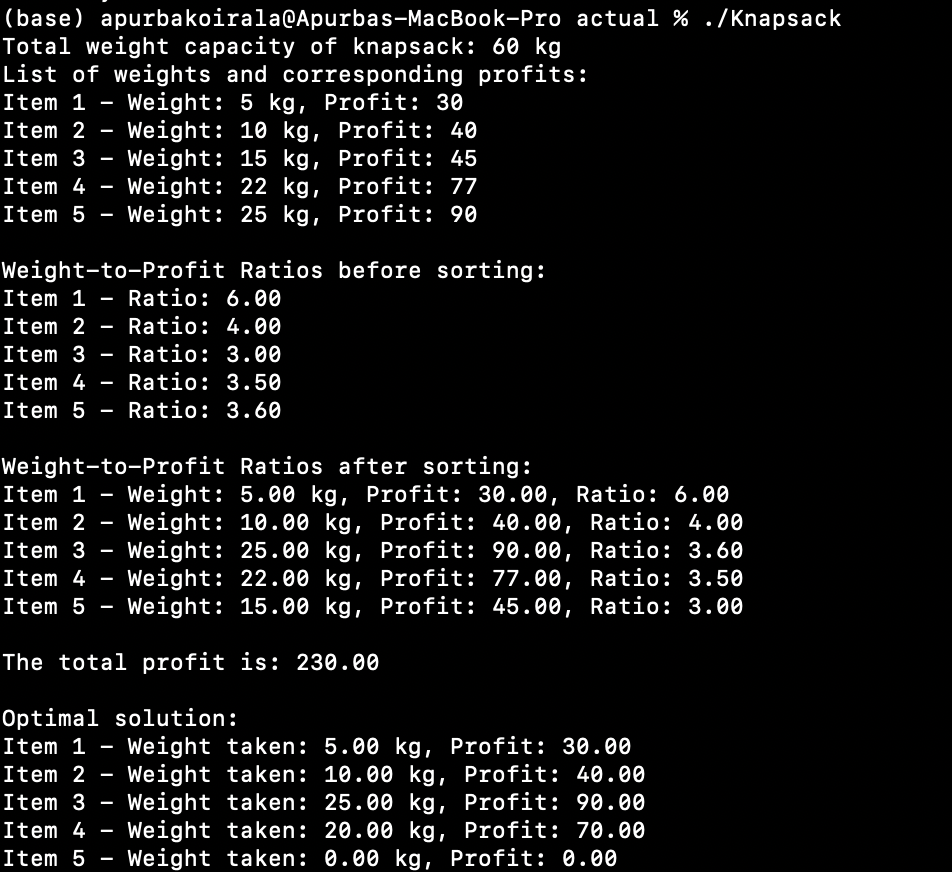
}

Input:

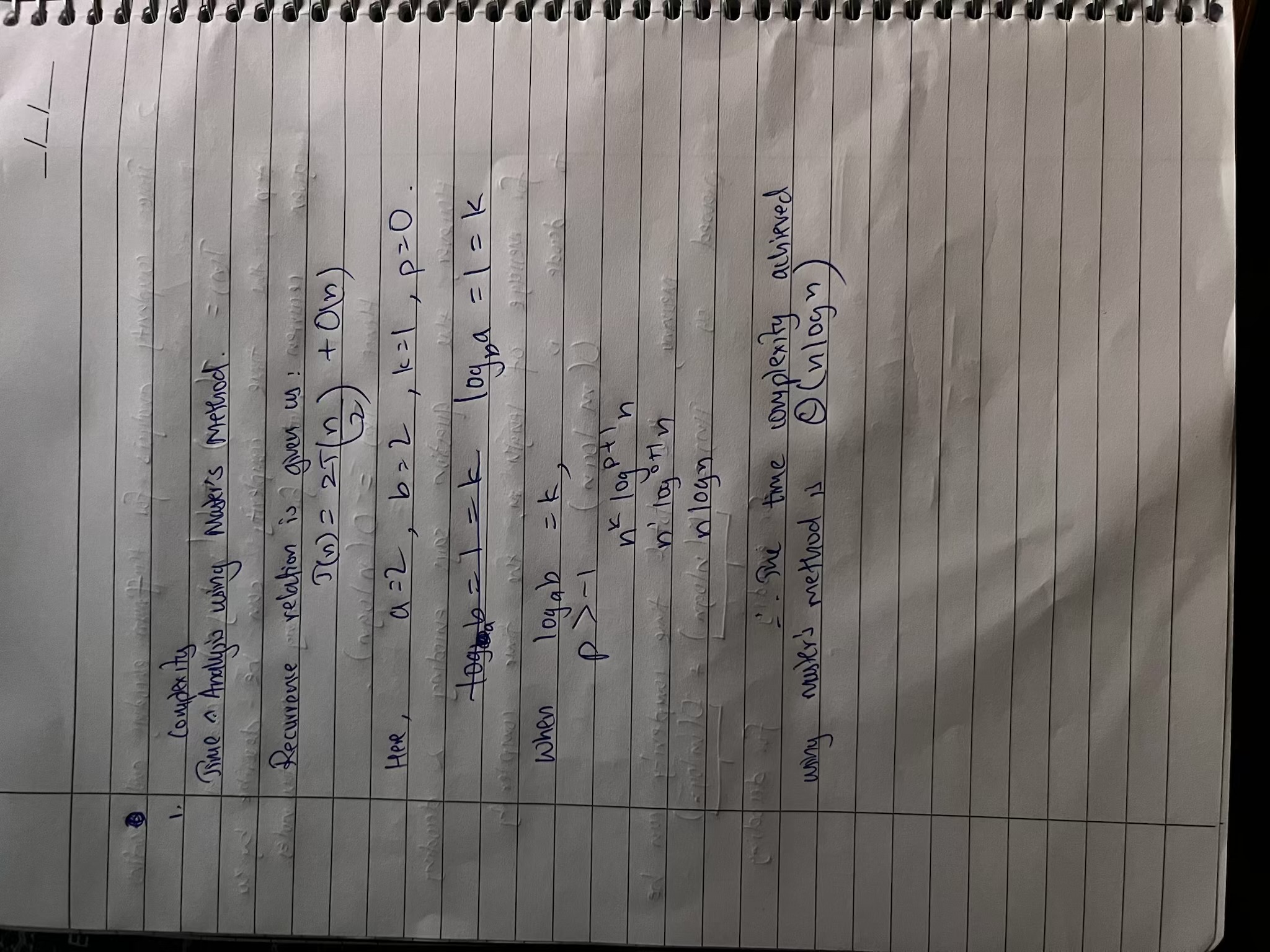
double P[] = {30, 40, 45, 77, 90};

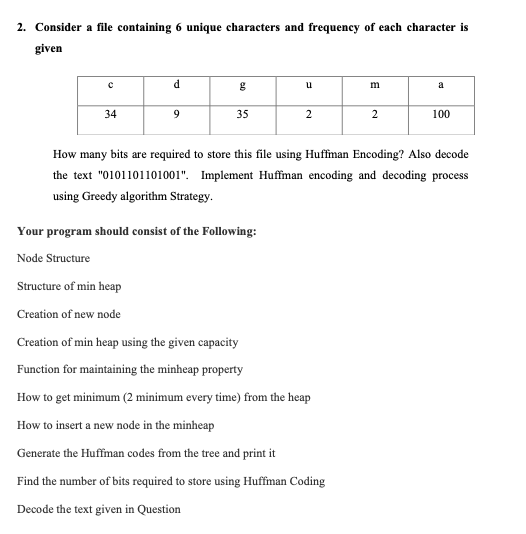
double W[] = {5, 10, 15, 22, 25};

Output:



Time Complexity:





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:

A screen shot of a computer

Description automatically generated

Time complexity analysis:

A notebook with writing on it

Description automatically generated

A screenshot of a computer

Description automatically generated

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;

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:

A screenshot of a computer program

Description automatically generated

Time complexity analysis:

A notebook with writing on it

Description automatically generated

A white paper with black text

Description automatically generated

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:

#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:

A black background with white text

Description automatically generated

Time complexity analysis:

