Fib. Practical 1
********** Recursive Approach **********

```
#include<iostream>
using namespace std;
int fibonacciRecursive(int n) {
    if (n <= 1) {
        return n;
    }
    int number=fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
    return number;
}

int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;
    cout << "Fibonacci number at position " << n << " is: " <<
fibonacciRecursive(n) << endl;
    return 0;
}
```

######## Non Recursive Approach (Iterative Method) ######

```
#include<iostream>
using namespace std;
int fibonacciNonRecursive(int n) {
    if (n <= 1) {
        return n;
    }
    int prev = 0;
    int current = 1;
    int result;
    for (int i = 2; i <= n; ++i) {
        result = prev + current;
        prev = current;
        current = result;
    }
    return result;
}
int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;
    cout << "Fibonacci number at position " << n << " is: " <<
fibonacciNonRecursive(n) << endl;
    return 0;
}
```

```
#include <iostream>
#include <queue>
#include <unordered_map>

using namespace std;

// Node structure for Huffman tree
struct Node {
    char data;
    int freq;
    Node* left;
    Node* right;
};

// Comparator for priority queue
struct Compare {
    bool operator()(Node* left, Node* right) {
        return left->freq > right->freq;
    }
};

// Function to build Huffman tree
Node* buildHuffmanTree(unordered_map<char, int>& freqMap) {
    priority_queue<Node*, vector<Node*>, Compare> pq;
    for (auto& pair : freqMap) {
        Node* newNode = new Node();
        newNode->data = pair.first;
        newNode->freq = pair.second;
        newNode->left = nullptr;
        newNode->right = nullptr;
        pq.push(newNode);
    }

    while (pq.size() > 1) {
        Node* left = pq.top();
        pq.pop();
        Node* right = pq.top();
        pq.pop();

        Node* internalNode = new Node();
        internalNode->data = '$';
        internalNode->freq = left->freq + right->freq;
        internalNode->left = left;
        internalNode->right = right;
        pq.push(internalNode);
    }

    return pq.top();
}
```

```
// Function to print Huffman codes
void printHuffmanCodes(Node* root, string code, unordered_map<char, string>&
huffmanCodes) {
    if (root == nullptr) {
        return;
    }

    if (root->data != '$') {
        huffmanCodes[root->data] = code;
    }

    printHuffmanCodes(root->left, code + "0", huffmanCodes);
    printHuffmanCodes(root->right, code + "1", huffmanCodes);
}

int main() {
    string input = "Huffman coding is a greedy algorithm";
    unordered_map<char, int> freqMap;

    // Calculate frequency of characters
    for (char ch : input) {
        if (ch != ' ') {
            freqMap[ch]++;
        }
    }

    // Build Huffman tree
    Node* root = buildHuffmanTree(freqMap);

    // Generate Huffman codes
    unordered_map<char, string> huffmanCodes;
    printHuffmanCodes(root, "", huffmanCodes);

    // Print Huffman codes
    cout << "Huffman Codes:" << endl;
    for (auto& pair : huffmanCodes) {
        cout << pair.first << ": " << pair.second << endl;
    }

    return 0;
}


----------

Output

/tmp/pXfMB8zzQN.o
Huffman Codes:
g: 1111
```

c: 11100
m: 1101
n: 1100
a: 000
i: 001
o: 0100
y: 01010
l: 10111
u: 01011
h: 10110
H: 11101
e: 0110
r: 1001
t: 01110
s: 01111
f: 1010
d: 1000

```
%%*** Knapsack ****

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Item {
    int weight;
    int value;
    double ratio; // Value-to-weight ratio
};

bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}

double fractionalKnapsack(vector<Item>& items, int capacity) {
    sort(items.begin(), items.end(), compare);
    double totalValue = 0.0;
    int currentWeight = 0;

    for (const auto& item : items) {
        if (currentWeight + item.weight <= capacity) {
            totalValue += item.value;
            currentWeight += item.weight;
        } else {
            int remainingWeight = capacity - currentWeight;
            totalValue += item.ratio * remainingWeight;
            break;
        }
    }

    return totalValue;
}

int main() {
    int n; // Number of items
    int capacity; // Knapsack capacity
    cout << "Enter the number of items: ";
    cin >> n;
    cout << "Enter the knapsack capacity: ";
    cin >> capacity;

    vector<Item> items(n);
    cout << "Enter weights and values of items:" << endl;
    for (int i = 0; i < n; ++i) {
        cin >> items[i].weight >> items[i].value;
        items[i].ratio = static_cast<double>(items[i].value) / items[i].weight;
    }
```

```
    double maxValue = fractionalKnapsack(items, capacity);
    cout << "Maximum value in knapsack = " << maxValue << endl;

    return 0;
}
```

OUTPUT

```
Enter the number of items: 3
Enter the knapsack capacity: 50
Enter weights and values of items:
10 60
20 100
30 120
Maximum value in knapsack = 240
```

****** Knap Sack 0 - 1 *********

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

int knapsack(int capacity, vector<int>& weights, vector<int>& values, int n) {
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= capacity; ++w) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w -
weights[i - 1]]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

int main() {
    int n; // Number of items
    int capacity; // Knapsack capacity
    cout << "Enter the number of items: ";
    cin >> n;
    cout << "Enter the knapsack capacity: ";
    cin >> capacity;

    vector<int> weights(n);
    vector<int> values(n);

    cout << "Enter weights of items:" << endl;
    for (int i = 0; i < n; ++i) {
        cin >> weights[i];
    }

    cout << "Enter values of items:" << endl;
    for (int i = 0; i < n; ++i) {
        cin >> values[i];
    }

    int maxValue = knapsack(capacity, weights, values, n);
    cout << "Maximum value in knapsack = " << maxValue << endl;

    return 0;
```

```
}
```

Output

```
Enter the number of items: 3
Enter the knapsack capacity: 2
Enter weights of items:
2
3
4
4
22
Enter values of items:
2
4
4
2
4
42
Maximum value in knapsack = 2
```

******** Quick Sort ******

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
using namespace std;
using namespace chrono;

// Function to partition the array for Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Deterministic Quick Sort
void quickSortDeterministic(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSortDeterministic(arr, low, pi - 1);
        quickSortDeterministic(arr, pi + 1, high);
    }
}

// Randomized Quick Sort
int randomPartition(int arr[], int low, int high) {
    srand(time(0));
    int random = low + rand() % (high - low);
    swap(arr[random], arr[high]);
    return partition(arr, low, high);
}

void quickSortRandomized(int arr[], int low, int high) {
    if (low < high) {
        int pi = randomPartition(arr, low, high);
        quickSortRandomized(arr, low, pi - 1);
        quickSortRandomized(arr, pi + 1, high);
    }
}
```

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

    int* arrDet = new int[n];
    int* arrRand = new int[n];

    // Filling arrays with random numbers
    for (int i = 0; i < n; i++) {
        arrDet[i] = arrRand[i] = rand() % 1000;
    }

    // Analyzing Deterministic Quick Sort
    auto startDet = high_resolution_clock::now();
    quickSortDeterministic(arrDet, 0, n - 1);
    auto stopDet = high_resolution_clock::now();
    auto durationDet = duration_cast<microseconds>(stopDet - startDet);

    // Analyzing Randomized Quick Sort
    auto startRand = high_resolution_clock::now();
    quickSortRandomized(arrRand, 0, n - 1);
    auto stopRand = high_resolution_clock::now();
    auto durationRand = duration_cast<microseconds>(stopRand - startRand);

    cout << "Deterministic Quick Sort Time: " << durationDet.count() << "
microseconds" << endl;
    cout << "Randomized Quick Sort Time: " << durationRand.count() << "
microseconds" << endl;

    delete[] arrDet;
    delete[] arrRand;

    return 0;
}
```

Output

```
Enter the size of the array: 4
Deterministic Quick Sort Time: 0 microseconds
Randomized Quick Sort Time: 9 microseconds
```