1. Write iterative algorithm for the following problem, write program in C/C++. Also analyze their complexity.

- **a. Greatest Common Divisor (GCD) of two numbers.**
- **b. Find nth Fibonacci number**
- **c. Sequential Search**
- **d. Bubble Sort,**
- **e. Selection Sort,**
- **f. Insertion Sort**
- **g. Min and Max finding in a list(min-max algorithm)**

Solution:

a. Greatest Common Divisor (GCD) of two numbers:

ALGORITHM:

1. Initialize two variables num1 and num2 with the given two numbers.
2. Repeat until num2 becomes 0:
    a. Compute the remainder by dividing num1 by num2.
    b. Set num1 to num2.
    c. Set num2 to remainder.
3. Return num1 as the GCD of the given two numbers.

SOURCE CODE:

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

int gcd(int num1, int num2) {
   int remainder;
   while (num2 != 0) {
      remainder = num1 % num2;
      num1 = num2;
      num2 = remainder;
   }
   return num1;
}

int main() {
   int num1 = 56, num2 = 84;
   int result = gcd(num1, num2);
   cout << "GCD of " << num1 << " and " << num2 << " is " << result << endl;
   return 0;
}
```

OUTPUT:

```
GCD of 56 and 84 is 28
```

COMPLEXITY:
Time Complexity: O(n)
Space Complexity: O(1)

b. Find nth Fibonacci number:

ALGORITHM:

1. Initialize two variables a and b as 0 and 1.
2. Repeat n-1 times:
    a. Set temp = b.
    b. Set b = a + b.
    c. Set a = temp.
3. Return a as the nth Fibonacci number.

SOURCE CODE:

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

int fibonacci(int n) {
   int a = 0, b = 1, temp;
   for (int i = 1; i < n; i++) {
      temp = b;
      b = a + b;
      a = temp;
   }
   return a;
}

int main() {
   int n = 7;
   int result = fibonacci(n);
   cout << "The " << n << "th Fibonacci number is " << result << endl;
   return 0;
}
```

OUTPUT:

The 7th Fibonacci number is 8

COMPLEXITY:
Time complexity: O(n) because the loop runs n-1 times.
Space complexity: O(1)

c. Sequential Search:

ALGORITHM:

1. Initialize i as 0.
2. Repeat until i becomes equal to n or arr[i] is equal to key:
    a. If arr[i] is equal to key, return i.
    b. Otherwise, increment i.
3. Return -1 as key is not found in the array.

SOURCE CODE:

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

int sequentialSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    int arr[] = {2, 4, 1, 7, 5};
    int n = 5;
    int key = 7;
    int result = sequentialSearch(arr, n, key);
    if (result == -1) {
        cout << "Key not found in the array" << endl;
    } else {
        cout << "Key found at index " << result << endl;
    }
    return 0;
}
```

OUTPUT:

Key found at index 3

COMPLEXITY:

Time complexity: O(n) in the worst case when the key is not present in the array.
Space complexity: O(n) as arr[ ] takes n memory references.

d. Bubble Sort:

ALGORITHM:

1. Repeat n-1 times:
    a. Initialize a flag swapped as false.
    b. Repeat from i=0 to n-2:
        i. If arr[i] is greater than arr[i+1], swap arr[i] and arr[i+1] and set swapped as true.
        ii. If swapped is still false, break the loop as the array is already sorted.
2. Return the sorted array.

SOURCE CODE:

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

void bubbleSort(int arr[], int n) {
   bool swapped;
   for (int i = 0; i < n-1; i++) {
      swapped = false;
      for (int j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            swap(arr[j], arr[j+1]);
            swapped = true;
         }
      }
      if (swapped == false) {
         break;
      }
   }
}

int main() {
   int arr[] = {2, 4, 1, 7, 5};
   int n = 5;
   cout << "Original array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;
   bubbleSort(arr, n);
   cout << "Sorted array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;
```

```
    return 0;
}
```

OUTPUT:

Original array: 2 4 1 7 5
Sorted array: 1 2 4 5 7

COMPLEXITY:

Time complexity: $O(n^2)$ in the average and worst case when the array is reverse sorted. The best case time complexity is O(n).
Space complexity: O(n)

e. Selection Sort:

ALGORITHM:

1. Repeat n-1 times:
    a. Initialize minIndex as i.
    b. Repeat from i+1 to n-1:
        i. If arr[j] is less than arr[minIndex], set minIndex as j.
        ii. If minIndex is not equal to i, swap arr[minIndex] and arr[i].
2. Return the sorted array.

SOURCE CODE:

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

void selectionSort(int arr[], int n) {
    int minIndex;
    for (int i = 0; i < n-1; i++) {
        minIndex = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        if (minIndex != i) {
            swap(arr[minIndex], arr[i]);
        }
    }
}

int main() {
    int arr[] = {2, 4, 1, 7, 5};
    int n = 5;
```

```
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    selectionSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

OUTPUT:

Original array: 2 4 1 7 5
Sorted array: 1 2 4 5 7

COMPLEXITY:

Time complexity: $O(n^2)$ in the worst case when the array is reverse sorted.
Space complexity: O(n)

f. Insertion Sort:

ALGORITHM:

1. Repeat from i=1 to n-1:
    a. Set key as arr[i].
    b. Repeat from j=i-1 to 0:
        i. If arr[j] is greater than key, shift arr[j] to the right.
        ii. If arr[j] is less than or equal to key, break the loop.
    c. Set arr[j+1] as key.
2. Return the sorted array.

SOURCE CODE:

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

void insertionSort(int arr[], int n) {
    int key, j;
    for (int i = 1; i < n; i++) {
        key = arr[i];
        j = i-1;
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
```

```
            j--;
        }
        arr[j+1] = key;
    }
}

int main() {
    int arr[] = {2, 4, 1, 7, 5};
    int n = 5;
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    insertionSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

OUTPUT:

Original array: 2 4 1 7 5
Sorted array: 1 2 4 5 7

COMPLEXITY:

Time complexity: $O(n^2)$ in the worst case when the array is reverse sorted. However, it performs better than bubble sort and selection sort when the array is partially sorted.
Space complexity: O(n)

g. Min and Max finding in a list(min-max algorithm):

ALGORITHM:

1. Initialize min and max as the first element of the array.
2. Repeat from i=2 to n:
    a. If arr[i] is greater than max, set max as arr[i].
    b. If arr[i] is less than min, set min as arr[i].
3. Return min and max.

SOURCE CODE:

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

```
void findMinMax(int arr[], int n, int& min, int& max) {
    min = max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        } else if (arr[i] < min) {
            min = arr[i];
        }
    }
}

int main() {
    int arr[] = {2, 4, 1, 7, 5};
    int n = 5;
    int min, max;
    findMinMax(arr, n, min, max);
    cout << "Minimum: " << min << endl;
    cout << "Maximum: " << max << endl;
    return 0;
}
```

OUTPUT:

Minimum: 1
Maximum: 7

COMPLEXITY:

Time complexity: O(n) since the array is iterated only once.

2. Write algorithm and program to search an item in ordered list using binary search as divide and conquer Approach.

Solution:

ALGORITHM:

1. Get the item to search and the ordered list to search it in.
2. Set the starting index to 0 and the ending index to the last index of the list.
3. While the starting index is less than or equal to the ending index, do the following:
    a. Calculate the middle index by adding the starting index and the ending index, and then dividing the result by 2 (integer division).
    b. If the middle item is equal to the item to search, return the middle index.
    c. If the middle item is less than the item to search, update the starting index to the middle index plus one.

    d. If the middle item is greater than the item to search, update the ending index to the middle index minus one.
4. If the item is not found, return -1.

SOURCE CODE:

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

int binary_search(int list[], int size, int item) {
    int start = 0;
    int end = size - 1;

    while (start <= end) {
        int middle = (start + end) / 2;

        if (list[middle] == item) {
            return middle;
        } else if (list[middle] < item) {
            start = middle + 1;
        } else {
            end = middle - 1;
        }
    }

    return -1;
}

int main() {
    int list[] = {2, 5, 8, 10, 13, 15, 18, 20};
    int size = sizeof(list) / sizeof(list[0]);
    int item = 13;

    int index = binary_search(list, size, item);

    if (index == -1) {
        cout << "Item not found" << endl;
    } else {
        cout << "Item found at index " << index << endl;
    }

    return 0;
}
```

OUTPUT:
Item found at index 4

3. Write algorithm and program to find the maximum and minimum element in a list of numbers using dynamic programming approach.
Solution:

ALGORITHM:

1. Initialize min and max as the first element of the array.
2. Repeat from i=2 to n:
    a. If arr[i] is greater than max, set max as arr[i].
    b. If arr[i] is less than min, set min as arr[i].
3. Return min and max.

SOURCE CODE:

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

void findMinMax(int arr[], int n, int& min, int& max) {
    int dp[n+1][2]; // dp[i][0] represents minimum and dp[i][1] represents maximum
    dp[0][0] = dp[0][1] = arr[0];
    for (int i = 1; i < n; i++) {
        dp[i][0] = min(dp[i-1][0], arr[i]);
        dp[i][1] = max(dp[i-1][1], arr[i]);
    }
    min = dp[n-1][0];
    max = dp[n-1][1];
}

int main() {
    int arr[] = {2, 4, 1, 7, 5};
    int n = 5;
    int min, max;
    findMinMax(arr, n, min, max);
    cout << "Minimum: " << min << endl;
    cout << "Maximum: " << max << endl;
    return 0;
}
```

OUTPUT:

```
Minimum: 1
Maximum: 7
```

4. Write algorithms and program using divide and conquer approach for following sorting algorithms

a. Heap Sort
b. Quick Sort
c. Randomized Quick Sort
d. Merge Sort

Solution:

a. Heap Sort:

ALGORITHM:

1. Build a max heap from the array.
    a. Starting at the last non-leaf node, heapify each sub-tree rooted at that node. This can be done recursively.
    b. The sub-tree rooted at a node can be heapified by swapping the node with its largest child if the largest child is greater than the node, and then recursively heapifying the sub-tree rooted at the child node.
2. Swap the first and last elements of the heap.
3. Remove the last element (which is now the largest) from the heap and place it in the sorted part of the array.
4. Repeat steps 2-3 until the entire array is sorted.

SOURCE CODE:

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

void heapify(int arr[], int n, int i) {
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;

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

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

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

      heapify(arr, n, largest);
   }
}

void heapSort(int 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);
    }
}

int main() {
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is: \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

OUTPUT:

```
Sorted array is:
5 6 7 11 12 13
```

b. Quick Sort:

ALGORITHM:

1. Choose a pivot element from the array.
2. Partition the array into two sub-arrays based on the pivot element, such that all elements less than or equal to the pivot element are in the left sub-array, and all elements greater than the pivot element are in the right sub-array.
3. Recursively apply steps 1 and 2 to the left and right sub-arrays until each sub-array has only one element.

SOURCE CODE:

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

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);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    cout << "Sorted array is: \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
}
```

OUTPUT:

Sorted array is:
1 5 7 8 9 10

c. Randomized Quick Sort:

ALGORITHM:

1. Choose a pivot element from the array at random.
2. Partition the array into two sub-arrays based on the pivot element, such that all elements less than or equal to the pivot element are in the left sub-array, and all elements greater than the pivot element are in the right sub-array.

3. Recursively apply steps 1 and 2 to the left and right sub-arrays until each sub-array has only one element.

SOURCE CODE:

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

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);
}

int randomPartition(int arr[], int low, int high) {
    int n = high - low + 1;
    int pivot = rand() % n;
    swap(arr[low + pivot], arr[high]);
    return partition(arr, low, high);
}

void randomizedQuickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = randomPartition(arr, low, high);

        randomizedQuickSort(arr, low, pi - 1);
        randomizedQuickSort(arr, pi + 1, high);
    }
}

int main() {
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);

    randomizedQuickSort(arr, 0, n - 1);
```

```
    cout << "Sorted array is: \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

OUTPUT:

Sorted array is:
1 5 7 8 9 10

d. Merge Sort:

ALGORITHM:

1. Divide the input array into two halves.
2. Recursively apply steps 1-2 to each half until each half has only one element.
3. Merge the two sorted halves to produce a sorted array.

SOURCE CODE:

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

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
```

```
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l >= r) {
        return;
    }
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
}

int main() {
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    mergeSort(arr, 0, n - 1);
    cout << "Sorted array is: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

OUTPUT:

Given array is: 12 11 13 5 6 7

Sorted array is: 5 6 7 11 12 13

5. Write a program to implement problem Job Sequencing with deadline using greedy algorithm.

Solution:

SOURCE CODE:

```
#include <iostream>
#include <algorithm>

using namespace std;

// Job structure
struct Job {
   int id;     // Job ID
   int deadline;   // Job Deadline
   int profit;     // Job Profit
};

// Function to compare jobs based on profit
bool compareJobs(Job a, Job b) {
   return (a.profit > b.profit);
}

// Function to find the maximum deadline in the given jobs array
int findMaxDeadline(Job jobs[], int n) {
   int max_deadline = 0;
   for(int i=0; i<n; i++) {
      max_deadline = max(max_deadline, jobs[i].deadline);
   }
   return max_deadline;
}

// Function to find the maximum profit using greedy approach
void findMaxProfit(Job jobs[], int n) {
   // Sort jobs based on profit in non-increasing order
   sort(jobs, jobs+n, compareJobs);

   // Find maximum deadline in the jobs array
   int max_deadline = findMaxDeadline(jobs, n);

   // Create an array to store the selected jobs
   int selected_jobs[max_deadline] = {0};

   // Iterate through the jobs array
```

```
    for(int i=0; i<n; i++) {
        // Find the latest available slot for the current job
        for(int j=jobs[i].deadline-1; j>=0; j--) {
            if(selected_jobs[j] == 0) {
                selected_jobs[j] = jobs[i].id;
                break;
            }
        }
    }

    // Print the selected jobs and their total profit
    int total_profit = 0;
    cout << "Selected Jobs: ";
    for(int i=0; i<max_deadline; i++) {
        if(selected_jobs[i] != 0) {
            cout << selected_jobs[i] << " ";
            total_profit += jobs[selected_jobs[i]-1].profit;
        }
    }
    cout << "\nTotal Profit: " << total_profit;
}

int main() {
    // Input the number of jobs
    int n;
    cout << "Enter the number of jobs: ";
    cin >> n;

    // Create an array to store the job information
    Job jobs[n];

    // Input the job information
    for(int i=0; i<n; i++) {
        cout << "Enter the deadline and profit for Job " << i+1 << ": ";
        cin >> jobs[i].deadline >> jobs[i].profit;
        jobs[i].id = i+1;
    }

    // Find the maximum profit using greedy approach
    findMaxProfit(jobs, n);

    return 0;
}
```

OUTPUT:

```
Enter the number of jobs: 5
Enter the deadline and profit for Job 1: 2 60
Enter the deadline and profit for Job 2: 1 100
Enter the deadline and profit for Job 3: 3 20
Enter the deadline and profit for Job 4: 2 40
Enter the deadline and profit for Job 5: 1 20

Selected Jobs: 2 1 4
Total Profit: 200
```

6. Implement the Floyd- Warshall's algorithm to compute all pair shortest path problem using dynamic programming approach.
Solution:

SOURCE CODE:

```
#include <iostream>

using namespace std;

const int INF = 1e9;

// Function to compute the shortest path between all pairs of vertices using Floyd-Warshall algorithm
void floydWarshall(int graph[][100], int n) {
    // Create a 2D array to store the shortest distances
    int dist[n][n];

    // Initialize the distance array with the graph matrix
    for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    // Compute the shortest distance between all pairs of vertices
    for(int k=0; k<n; k++) {
        for(int i=0; i<n; i++) {
            for(int j=0; j<n; j++) {
                if(dist[i][k] != INF && dist[k][j] != INF && dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
```

```
            }
        }
    }
}

// Print the shortest distances
cout << "Shortest Distances between all pairs of vertices:\n";
for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) {
        if(dist[i][j] == INF) {
            cout << "INF ";
        }
        else {
            cout << dist[i][j] << " ";
        }
    }
    cout << "\n";
}
}

int main() {
// Input the number of vertices and edges
int n, m;
cout << "Enter the number of vertices and edges: ";
cin >> n >> m;

// Create a 2D array to store the graph information
int graph[100][100];

// Initialize the graph matrix with maximum values
for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) {
        graph[i][j] = INF;
    }
}

// Input the graph information
int u, v, w;
for(int i=0; i<m; i++) {
    cout << "Enter the source, destination, and weight of edge " << i+1 << ": ";
    cin >> u >> v >> w;
    graph[u-1][v-1] = w;
}
```

```
    // Compute the shortest path between all pairs of vertices using Floyd-Warshall algorithm
    floydWarshall(graph, n);

    return 0;
}
```

OUTPUT:

```
Enter the number of vertices and edges: 4 5
Enter the source, destination, and weight of edge 1: 1 2 3
Enter the source, destination, and weight of edge 2: 1 3 8
Enter the source, destination, and weight of edge 3: 2 4 2
Enter the source, destination, and weight of edge 4: 3 2 5
Enter the source, destination, and weight of edge 5: 3 4 6

Shortest Distances between all pairs of vertices:
INF 3 8 5
INF INF INF 2
INF 5 INF 6
INF INF INF INF
```

7. Write a program to implement Matrix Chain Multiplication using dynamic programming algorithm.

Solution:

SOURCE CODE:

```
#include <iostream>
#include <limits.h>

using namespace std;

int matrixChainMultiplication(int dims[], int n) {
    int m[n][n];

    for(int i=0; i<n; i++) {
        m[i][i] = 0;
    }

    for(int l=2; l<=n; l++) {
        for(int i=0; i<n-l+1; i++) {
            int j = i+l-1;
            m[i][j] = INT_MAX;
            for(int k=i; k<j; k++) {
```

```
                int q = m[i][k] + m[k+1][j] + dims[i]*dims[k+1]*dims[j+1];
                if(q < m[i][j]) {
                    m[i][j] = q;
                }
            }
        }
    }

    return m[0][n-2];
}

int main() {
    int n;
    cout << "Enter the number of matrices: ";
    cin >> n;
    int dims[n+1];
    cout << "Enter the dimensions of the matrices: ";
    for(int i=0; i<=n; i++) {
        cin >> dims[i];
    }

    int minMult = matrixChainMultiplication(dims, n+1);

    cout << "Minimum number of multiplications needed to multiply the sequence of matrices: "
<< minMult << endl;

    return 0;
}
```

OUTPUT:

```
Enter the number of matrices: 3
Enter the dimensions of the matrices: 10 20 30 40
Minimum number of multiplications needed to multiply the sequence of matrices: 18000
```