

Lab Report 1: Design and Analysis of GCD Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

The **Greatest Common Divisor (GCD)** of two integers a and b is the largest integer that divides both a and b without leaving a remainder. It is also known as the *highest common factor (HCF)*. The GCD has applications in number theory, cryptography, and computational mathematics.

Key Concepts:

- If $\gcd(a, b) = 1$, then a and b are said to be *coprime*.
- The Euclidean algorithm is one of the most efficient methods to compute the GCD by repeatedly applying division.

Properties of GCD:

1. $\gcd(a, 0) = |a|$.
2. $\gcd(a, b) = \gcd(b, a \bmod b)$, where **mod** represents the remainder operation.
3. The algorithm terminates when the remainder becomes zero.

2. Algorithm:

Euclidean Algorithm for GCD:

The Euclidean algorithm computes the GCD by iteratively reducing the problem size using division with remainder. Below is the pseudocode for the algorithm:

Algorithm: Euclidean_GCD (a, b)
Input: Two positive integers a and b ($a \geq b > 0$)
Output: The greatest common divisor of a and b .
1. If $b=0$, return a as the GCD.
2. Otherwise: <ul style="list-style-type: none">• Compute $r = a \bmod b$ (remainder when a is divided by b).• Replace a with b, and b with r.• Repeat Step 1 until $b = 0$.
3. Return a as the GCD.

3. Time and Space Complexity Analysis:

Time Complexity: $O(n)$

Space Complexity: $O(1)$

4. Code Implementation:

```
/*Program that calculates the GCD of given N numbers*/
#include<iostream>
using namespace std;

// Function to calculate GCD of two numbers
int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

// Function to calculate GCD of an array of numbers
int findGCD(int arr[], int n) {
    int result = arr[0];
    for (int i = 1; i < n; i++) {
        result = gcd(result, arr[i]);
        if(result == 1) {
            return 1;
        }
    }
    return result;
}

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "GCD of the given numbers is: " << findGCD(arr, n) <<
endl;
    return 0;
}
```

5. Output:

```
■ > ... > College > main ?3 ~5 -5 14:54 0.065s cd "c:\Users\firoj\OneDrive\
-o GCD } ; if ($?) { .\GCD }
• Enter the number of elements: 2
Enter the elements: 8
16
GCD of the given numbers is: 8
```

Lab Report 2: Design and Analysis of Factorial Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

The factorial of a non-negative integer n , denoted as $n!$, is the product of all positive integers less than or equal to n . For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Factorial has applications in combinatorics, probability, and algorithm design. The base case is $0! = 1$ by definition. Factorial can be computed iteratively or recursively, with recursion being a common approach in programming.

2. Key Concepts:

- Factorial grows extremely fast, leading to large numbers even for small inputs.
- Recursive definition: $n! = n \times (n - 1)!$, with base case $0! = 1$.
- Iterative solutions avoid stack overflow for large n , unlike recursion.

3. Properties of Factorial:

- $n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$.
- Factorial is undefined for negative numbers.
- For large n , factorial exceeds typical integer limits, requiring special handling (e.g., long or big integer types).

4. Algorithm:

Recursive Algorithm for Factorial:

The recursive factorial algorithm computes $n!$ by breaking it into smaller subproblems. Below is the algorithm:

Algorithm: Factorial (n)

Input: A non-negative integer n ($n \geq 0$)

Output: The factorial of n

1. If $n = 0$, return **1** as the factorial.
 2. Otherwise:
 - Compute $n \times \text{Factorial}(n - 1)$.
 - Return the result.
-

5. Time and Space Complexity Analysis:

Time Complexity: $O(n)$ – The algorithm makes n recursive calls or iterations.

Space Complexity: $O(n)$ – For the recursive version, the call stack uses space proportional to n . (Iterative version would be $O(1)$.)

6. Code Implementation:

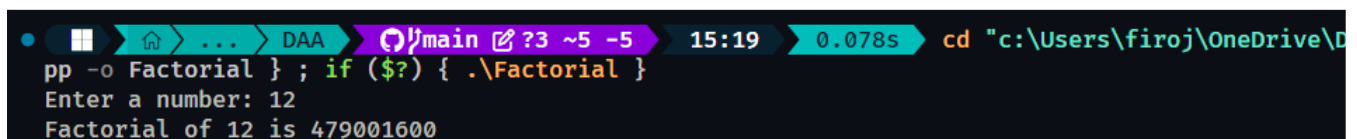
```
/*The program for factorial of given N numbers*/

#include<iostream>
using namespace std;

// Function for factorial
int factorial(int n) {
    if(n <= 1) return 1;
    else return n * factorial(n - 1);
}

int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;
    cout << "Factorial of " << n << " is " << factorial(n) << endl;
    return 0;
}
```

7. Output:



```
• [Taskbar icons] DAA [main] ?3 ~5 -5 15:19 0.078s cd "c:\Users\firoj\OneDrive\
pp -o Factorial } ; if ($?) { .\Factorial }
Enter a number: 12
Factorial of 12 is 479001600
```

Lab Report 3: Design and Analysis of Fibonacci Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, typically starting with 0 and 1. Formally, it is defined as: $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$. The sequence begins: 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on. The Fibonacci sequence has applications in mathematics, computer science, and nature (e.g., modeling growth patterns).

2. Key Concepts:

- The sequence can be computed recursively or iteratively.
- Recursive computation is intuitive but inefficient for large n due to exponential growth in calls.
- Iterative computation is more efficient, using constant extra space and linear time.

3. Properties of Fibonacci:

- $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$.
- The ratio of consecutive Fibonacci numbers approximates the golden ratio (~ 1.618) as n increases.
- Fibonacci numbers grow exponentially, requiring careful handling for large n (e.g., using long or big integer types).

4. Algorithm:

Iterative Algorithm for Fibonacci:

The iterative Fibonacci algorithm computes the n^{th} Fibonacci number by maintaining two variables and updating them in a loop. Below is the algorithm:

Algorithm: Fibonacci (n)

Input: A non-negative integer n : ($n \geq 0$)

Output: The n^{th} Fibonacci number

1. If $n = 0$, return **0**.

2. If $n = 1$, return **1**.

3. Otherwise:

 ○ Set $a = 0$ (first number), $b = 1$ (second number).

 ○ For $i = 2$ to n :

 ▪ Compute $c = a + b$.

 ▪ Update $a = b, b = c$.

 ○ Return b as the n^{th} Fibonacci number.

5. Time and Space Complexity Analysis:

Time Complexity: $O(n)$ – The algorithm performs $n - 1$ iterations to compute the n^{th} number.

Space Complexity: $O(1)$ – Only a constant amount of extra space is used (variables a , b , and c).

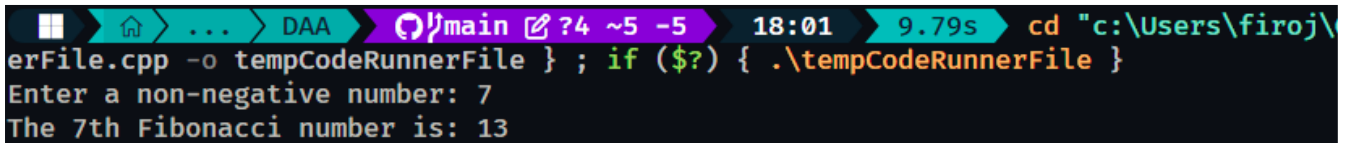
6. Code Implementation:

```
/*Program that calculates the nth Fibonacci number*/
#include<iostream>
using namespace std;

// Function to calculate the nth Fibonacci number
int fibonacci(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

int main() {
    int n;
    cout << "Enter a non-negative number: ";
    cin >> n;
    if (n < 0) {
        cout << "Fibonacci is not defined for negative numbers." <<
endl;
    } else {
        cout << "The " << n << "th Fibonacci number is: " <<
        fibonacci(n) << endl;
    }
    return 0;
}
```

7. Output:



```

> DAA > g++ main.cpp -o tempCodeRunnerFile
Enter a non-negative number: 7
The 7th Fibonacci number is: 13
```


Lab Report 4: Design and Analysis of Linear Search Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Linear search, also known as sequential search, is a simple algorithm used to find the position of a target element within a list or array. It works by examining each element in the array sequentially until the target is found or the end of the array is reached. Linear search is straightforward and does not require the data to be sorted, making it applicable to a wide range of problems.

2. Key Concepts:

- Linear search checks each element one by one, starting from the first index.
- It returns the index of the target element if found, or a sentinel value (e.g., -1) if not found.
- The algorithm's simplicity makes it useful for small datasets or unsorted arrays.

3. Properties of Linear Search:

- Works on both sorted and unsorted arrays.
- The search terminates early if the target is found; otherwise, it examines the entire array.
- Performance degrades as the size of the array increases.

4. Algorithm:

Linear Search Algorithm:

The linear search algorithm iterates through the array, comparing each element with the target value. Below is the pseudocode for the algorithm:

Algorithm: Linear Search (arr, n, target)

Input: An array arr of size n , and a target value target

Output: The index of target in arr, or -1 if not found

1. For $i = 0$ to $n - 1$:

 o If $arr[i] = target$, return i as the index.

2. If no match is found after the loop, return -1 .

5. Time and Space Complexity Analysis:

- Time Complexity:

- **Best Case:** $O(1)$ – Target is found at the first position.
- **Worst Case:** $O(n)$ – Target is at the last position or not present.
- **Average Case:** $O(n)$ – On average, half the array is searched.

- **Space Complexity:** $O(1)$ – Only a constant amount of extra space is used (e.g., loop variable).

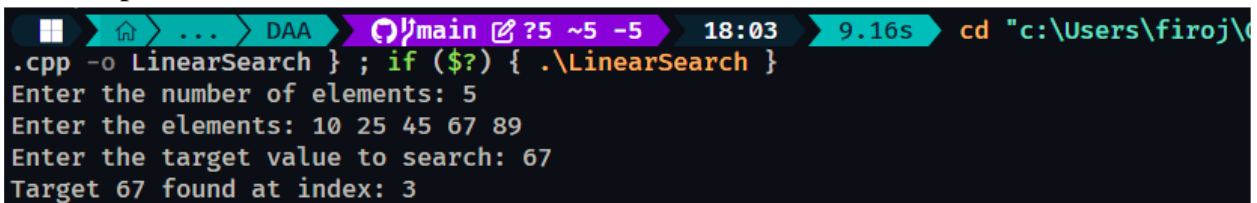
6. Code Implementation:

```
/*Program that performs linear search on an array*/
#include<iostream>
using namespace std;

// Function to perform linear search
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target)
            return i; // Return index if target is found
    }
    return -1; // Return -1 if target is not found
}

int main() {
    int n, target;
    cout << "Enter the number of elements: ";
    cin >> n;
    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "Enter the target value to search: ";
    cin >> target;
    int result = linearSearch(arr, n, target);
    if (result == -1) {
        cout << "Target " << target << " not found in the array." <<
endl;
    } else {
        cout << "Target " << target << " found at index: " << result
<< endl;
    }
    return 0;
}
```

8. Output:



```
main ?5 ~5 -5 18:03 9.16s cd "c:\Users\firoj\c
.cpp -o LinearSearch } ; if ($?) { .\LinearSearch }
Enter the number of elements: 5
Enter the elements: 10 25 45 67 89
Enter the target value to search: 67
Target 67 found at index: 3
```

Lab Report 5: Design and Analysis of Bubble Sort Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Bubble sort is a simple sorting algorithm that repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until no more swaps are needed, indicating that the list is sorted. It is called "bubble sort" because smaller elements "bubble" to the top (beginning) of the list in each iteration. Though inefficient for large datasets, it is easy to understand and implement.

2. Key Concepts:

- Bubble sort works by comparing and swapping adjacent elements in multiple passes.
- It can be optimized by checking if any swaps occurred in a pass; if not, the array is already sorted.
- It is a stable sorting algorithm, preserving the relative order of equal elements.

3. Properties of Bubble Sort:

- Best case occurs when the array is already sorted.
- Worst case occurs when the array is sorted in reverse order.
- The algorithm always performs pairwise comparisons, even if unnecessary swaps are avoided with optimization.

4. Algorithm:

Bubble Sort Algorithm:

The bubble sort algorithm iterates through the array, swapping adjacent elements if they are out of order, and repeats until the array is fully sorted. Below is the algorithm:

Algorithm: Bubble_Sort (arr, n)

Input: An array arr of size n

Output: The array arr sorted in ascending order

1. For i = 0 to n-1:

 ○ Set swapped = false.

 ○ For j = 0 to n-i-1:

 ▪ If arr[j] > arr[j+1]:

 ▪ Swap arr[j] and arr[j+1].

 ▪ Set swapped = true.

 ○ If swapped = false, break (array is sorted).

2. Return the sorted array.

5. Time and Space Complexity:

- Time Complexity:

- **Best Case:** $O(n)$ – Array is already sorted, and optimization detects no swaps in the first pass.
- **Worst Case:** $O(n^2)$ – Array is reverse sorted, requiring maximum comparisons and swaps.
- **Average Case:** $O(n^2)$ – Randomly ordered array requires roughly $n^2/4$ comparisons and swaps.

- **Space Complexity:** $O(1)$ – Only a constant amount of extra space is used (e.g., for temporary swaps).

6. Code Implementation

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

// Function to perform bubble sort and show the process
void bubbleSort(int arr[], int n) {
    int step = 1; // Initialize step counter
    // Traverse through all array elements
    for (int i = 0; i < n-1; i++) {
        // Last i elements are already in place
        for (int j = 0; j < n-i-1; j++) {
            // Print the elements being compared
            cout << "Comparing " << arr[j] << " and " << arr[j+1] <<
endl;

            // Swap if the element found is greater than the next
            element

            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
            // Print array after each swap
            cout << "Step " << step << ": ";
            for (int k = 0; k < n; k++)
                cout << arr[k] << " ";
            cout << endl;
            step++; // Increment step counter
        }
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
```

```

        cout << endl;
    }

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

        int arr[n];
        cout << "Enter the elements: ";
        for (int i = 0; i < n; i++) {
            cin >> arr[i];
        }

        // Print unsorted array
        cout << "Unsorted array: ";
        printArray(arr, n);

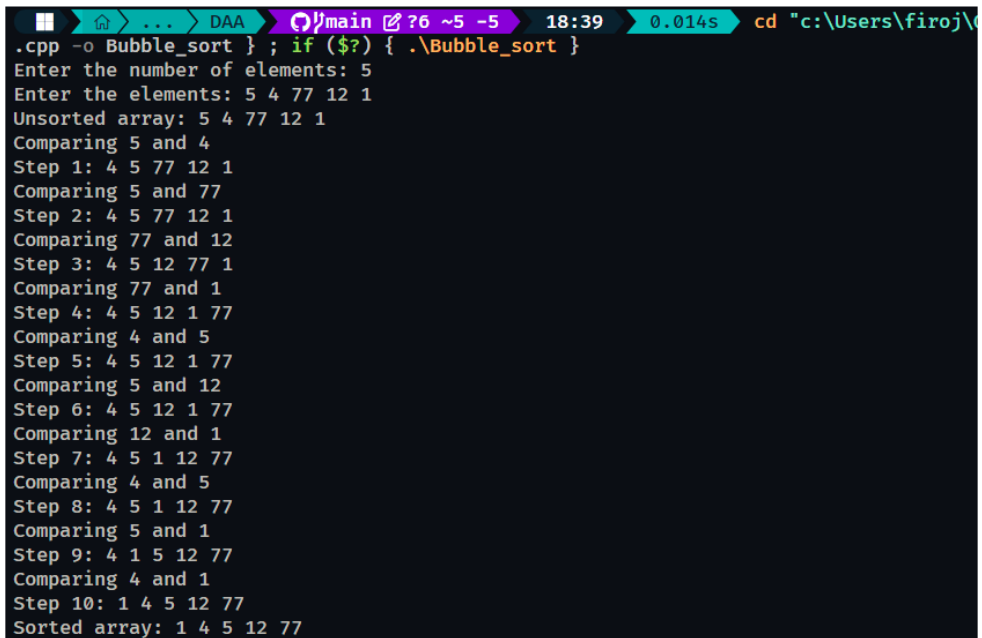
        // Perform bubble sort
        bubbleSort(arr, n);

        // Print sorted array
        cout << "Sorted array: ";
        printArray(arr, n);

        return 0;
    }

```

7. Output:



```

DAA main ?6 ~5 -5 18:39 0.014s cd "c:\Users\firoj\
.cpp -o Bubble_sort } ; if ($?) { .\Bubble_sort }
Enter the number of elements: 5
Enter the elements: 5 4 77 12 1
Unsorted array: 5 4 77 12 1
Comparing 5 and 4
Step 1: 4 5 77 12 1
Comparing 5 and 77
Step 2: 4 5 77 12 1
Comparing 77 and 12
Step 3: 4 5 12 77 1
Comparing 77 and 1
Step 4: 4 5 12 1 77
Comparing 4 and 5
Step 5: 4 5 12 1 77
Comparing 5 and 12
Step 6: 4 5 12 1 77
Comparing 12 and 1
Step 7: 4 5 1 12 77
Comparing 4 and 5
Step 8: 4 5 1 12 77
Comparing 5 and 1
Step 9: 4 1 5 12 77
Comparing 4 and 1
Step 10: 1 4 5 12 77
Sorted array: 1 4 5 12 77

```

Lab Report 6: Design and Analysis of Quick Sort Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Quick Sort is an efficient, comparison-based sorting algorithm that uses a divide-and-conquer strategy. It selects a "pivot" element from the array and partitions the other elements into two sub-arrays: those less than the pivot and those greater than the pivot. The sub-arrays are then recursively sorted. Quick Sort is widely used due to its average-case performance and in-place sorting capability, making it memory-efficient.

2. Key Concepts:

- The choice of pivot affects performance (e.g., first element, last element, random, or median).
- Partitioning rearranges the array so that the pivot is in its final sorted position.
- Quick Sort is not stable, meaning it may change the relative order of equal elements.

3. Properties of Quick Sort:

- Best case occurs when the pivot consistently divides the array into roughly equal halves.
- Worst case occurs with already sorted or reverse-sorted arrays and a poor pivot choice (e.g., first or last element).
- It performs in-place sorting, requiring minimal extra memory beyond the recursion stack.

4. Algorithm:

Quick Sort Algorithm:

The Quick Sort algorithm recursively partitions the array around a pivot. Below is the actual implementation in C++:

Algorithm: Quick_Sort (arr, low, high)

Input: An array arr, and indices low and high defining the sub-array to sort

Output: The array arr sorted in ascending order

1. If low < high:

- Call partition(arr, low, high) to get the pivot index pi.
 - Call Quick_Sort(arr, low, pi-1) to sort the left sub-array.
 - Call Quick_Sort(arr, pi+1, high) to sort the right sub-array.
-

2. Return the sorted array.

Algorithm: Partition (arr, low, high)

Input: An array arr, and indices low and high defining the sub-array

Output: The index of the pivot in its final sorted position

1. Set pivot = arr[high] (choose last element as pivot).
 2. Set i = low-1 (index of smaller element).
 3. For j = low to high-1:
 - If arr[j] <= pivot:
-

<ul style="list-style-type: none"> ▪ Increment i. ▪ Swap arr[i] and arr[j].
4. Swap arr[i+1] and arr[high] (place pivot in its final position).
5. Return i+1 as the pivot index.

5. Time and Space Complexity:

- Time Complexity:

- **Best Case:** $O(n \log n)$ – Pivot splits the array into two roughly equal halves.
- **Worst Case:** $O(n^2)$ – Pivot is the smallest or largest element, leading to unbalanced partitions.
- **Average Case:** $O(n \log n)$ – Random pivot selection typically yields balanced partitions.

- Space Complexity:

- $O(\log n)$ – Due to the recursion stack in the average and best cases.
- $O(n)$ – In the worst case, the recursion stack depth equals the array size.

6. Code Implementation:

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

// Function to partition the array and show the process
int partition(int arr[], int low, int high, int& step) {
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1;       // Index of smaller element
    for (int j = low; j < high; j++) {
        // Print the elements being compared
        cout << "Comparing " << arr[j] << " and " << pivot << endl;
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
        // Print array after each comparison
        cout << "Step " << step << ": ";
        for (int k = low; k <= high; k++)
            cout << arr[k] << " ";
        cout << endl;
        step++; // Increment step counter
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    // Print array after placing pivot
    cout << "Step " << step << " (Pivot " << pivot << " placed): ";
```

```

        for (int k = low; k <= high; k++)
            cout << arr[k] << " ";
        cout << endl;
        step++;
        return i + 1;
    }

// Function to perform Quick Sort and show the process
void quickSort(int arr[], int low, int high, int& step) {
    if (low < high) {
        int pi = partition(arr, low, high, step);
        quickSort(arr, low, pi - 1, step);
        quickSort(arr, pi + 1, high, step);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Print unsorted array
    cout << "Unsorted array: ";
    printArray(arr, n);

    // Perform Quick Sort
    int step = 1; // Initialize step counter
    quickSort(arr, 0, n-1, step);

    // Print sorted array
    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}

```


7. Output:

```

- o Quick_Sort } ; if ($?) { .\Quick_Sort }
Enter the number of elements: 5
Enter the elements: 44 2 14 56 22
Unsorted array: 44 2 14 56 22
Comparing 44 and 22
Step 1: 44 2 14 56 22
Comparing 2 and 22
Step 2: 2 44 14 56 22
Comparing 14 and 22
Step 3: 2 14 44 56 22
Comparing 56 and 22
Step 4: 2 14 44 56 22
Step 5 (Pivot 22 placed): 2 14 22 56 44
Comparing 2 and 14
Step 6: 2 14
Step 7 (Pivot 14 placed): 2 14
Comparing 56 and 44
Step 8: 56 44
Step 9 (Pivot 44 placed): 44 56
Sorted array: 2 14 22 44 56
```

Lab Report 7: Design and Analysis of Selection Sort Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Selection Sort is a simple comparison-based sorting algorithm that divides the input array into two parts: a sorted portion (initially empty) and an unsorted portion. In each iteration, it finds the minimum element in the unsorted portion and swaps it with the first element of the unsorted portion, effectively growing the sorted portion. It is inefficient for large datasets but easy to implement and understand.

2. Key Concepts:

- Selection Sort repeatedly selects the smallest (or largest) element from the unsorted section.
- It performs swaps only once per iteration, unlike Bubble Sort's frequent swaps.
- It is an in-place algorithm but not stable, as it may change the relative order of equal elements.

3. Properties of Selection Sort:

- The number of comparisons is always the same, regardless of the input order.
- Best case occurs when the array is already sorted (fewer swaps), but comparisons remain unchanged.
- Worst case occurs when the array is reverse sorted, maximizing swaps.

4. Algorithm:

Algorithm: Selection_Sort (arr, n)

Input: An array arr of size n

Output: The array arr sorted in ascending order

1. For $i = 0$ to $n - 1$:

- Set $\text{min_idx} = i$ (assume the current position holds the minimum).
-

- For $j = i + 1$ to $n - 1$:
-

- If $\text{arr}[j] < \text{arr}[\text{min_idx}]$:

- Set $\text{min_idx} = j$.

- If $\text{min_idx} \neq i$:
-

- Swap $\text{arr}[i]$ and $\text{arr}[\text{min_idx}]$.

2. Return the sorted array.

5. Time and Space Complexity Analysis:

- **Time Complexity:**
 - **Best Case:** $O(n^2)$ – Even if sorted, it performs all comparisons.
 - **Worst Case:** $O(n^2)$ – Comparisons and swaps are maximized.
 - **Average Case:** $O(n^2)$ – Roughly $n^2/2$ comparisons regardless of input.
- **Space Complexity:** $O(1)$ – Only a constant amount of extra space is used (e.g., for temporary variables).

6. Code Implementation:

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

// Function to perform selection sort and show the process
void selectionSort(int arr[], int n) {
    int step = 1;
    // Traverse through all array elements
    for (int i = 0; i < n-1; i++) {
        // Find the minimum element in unsorted array
        int min_idx = i;
        cout << "Step " << step << ": Initial minimum: " <<
arr[min_idx] << endl;
        for (int j = i+1; j < n; j++) {
            // Print the elements being compared
            cout << "Comparing " << arr[min_idx] << " and " << arr[j]
<< endl;
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
                cout << "New minimum found: " << arr[min_idx] << endl;
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;

        // Print array after each swap
        cout << "Array after step " << step << ": ";
        for (int k = 0; k < n; k++)
            cout << arr[k] << " ";
        cout << endl;
        step++; // Increment step counter
    }
}

// Function to print an array
void printArray(int arr[], int size) {
```

```

    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Print unsorted array
    cout << "Unsorted array: ";
    printArray(arr, n);

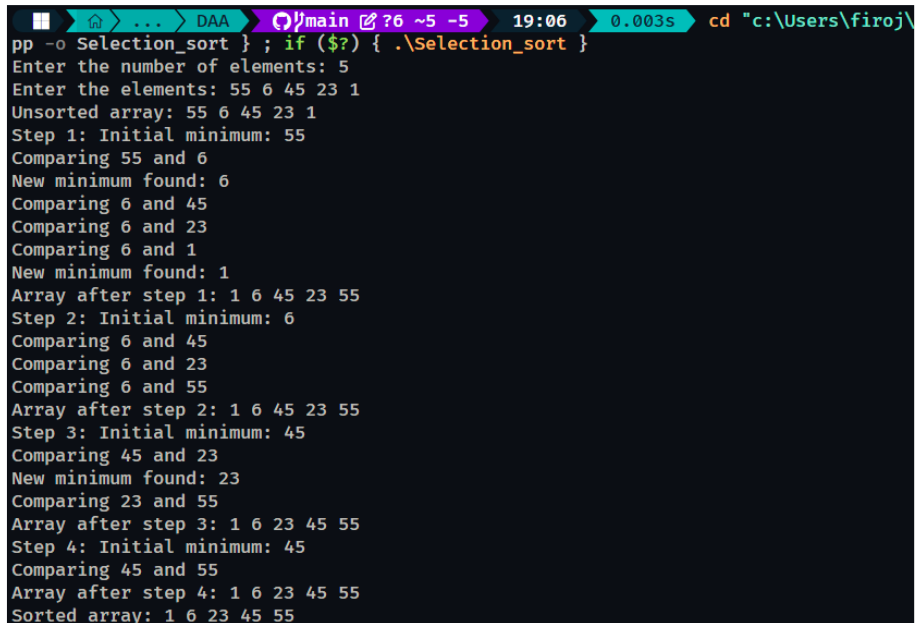
    // Perform selection sort
    selectionSort(arr, n);

    // Print sorted array
    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}

```

7. Output:



```

DAA main ?6 ~5 -5 19:06 0.003s cd "c:\Users\firoj\
pp -o Selection_sort } ; if ($?) { .\Selection_sort }
Enter the number of elements: 5
Enter the elements: 55 6 45 23 1
Unsorted array: 55 6 45 23 1
Step 1: Initial minimum: 55
Comparing 55 and 6
New minimum found: 6
Comparing 6 and 45
Comparing 6 and 23
Comparing 6 and 1
New minimum found: 1
Array after step 1: 1 6 45 23 55
Step 2: Initial minimum: 6
Comparing 6 and 45
Comparing 6 and 23
Comparing 6 and 55
Array after step 2: 1 6 45 23 55
Step 3: Initial minimum: 45
Comparing 45 and 23
New minimum found: 23
Comparing 23 and 55
Array after step 3: 1 6 23 45 55
Step 4: Initial minimum: 45
Comparing 45 and 55
Array after step 4: 1 6 23 45 55
Sorted array: 1 6 23 45 55

```

Lab Report 8: Design and Analysis of Insertion Sort Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Insertion Sort is a simple comparison-based sorting algorithm that builds a sorted portion of the array one element at a time. It takes each element from the unsorted portion and inserts it into its correct position within the sorted portion by shifting larger elements to the right. It mimics how people sort playing cards and is efficient for small datasets or nearly sorted arrays.

2. Key Concepts:

- Insertion Sort maintains a sorted sub-array that grows with each iteration.
- It shifts elements rather than swapping them, reducing write operations compared to other algorithms like Bubble Sort.
- It is stable (preserves the relative order of equal elements) and adaptive (faster for partially sorted data).

3. Properties of Insertion Sort:

- Best case occurs when the array is already sorted, requiring minimal shifts.
- Worst case occurs when the array is reverse sorted, maximizing comparisons and shifts.
- It works well for online sorting, where data arrives incrementally.

4. Algorithm:

Algorithm: Insertion_Sort (arr, n)

Input: An array arr of size n

Output: The array arr sorted in ascending order

1. For $i = 1$ to $n - 1$:

- Set $key = arr[i]$ (element to insert).
 - Set $j = i - 1$ (last index of sorted portion).
 - While $j \geq 0$ and $arr[j] > key$:
 - Shift $arr[j]$ to $arr[j+1]$.
 - Decrement j .
 - Place key at $arr[j+1]$.
-

2. Return the sorted array.

5. Time and Space Complexity Analysis:

- **Time Complexity:**
 - **Best Case:** $O(n)$ – Array is already sorted, requiring only one comparison per element.
 - **Worst Case:** $O(n^2)$ – Array is reverse sorted, requiring maximum comparisons and shifts.
 - **Average Case:** $O(n^2)$ – Roughly $n^2/4$ comparisons and shifts for random input.
- **Space Complexity:** $O(1)$ – Only a constant amount of extra space is used (e.g., for key and index variables).

6. Code Implementation:

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

// Function to perform Insertion Sort and show the process
void insertionSort(int arr[], int n) {
    int step = 1; // Initialize step counter
    // Traverse through all array elements starting from the second
    element
    for (int i = 1; i < n; i++) {
        int key = arr[i]; // Element to insert
        int j = i - 1;    // Last index of sorted portion
        cout << "Step " << step << ": Inserting key: " << key << endl;

        // Shift elements that are greater than key
        while (j >= 0 && arr[j] > key) {
            cout << "Comparing " << arr[j] << " and " << key << endl;
            cout << "Shifting " << arr[j] << " to the right" << endl;
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key; // Place key in its correct position

        // Print array after each insertion
        cout << "Array after step " << step << ": ";
        for (int k = 0; k < n; k++)
            cout << arr[k] << " ";
        cout << endl;
        step++; // Increment step counter
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

```

}

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

    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Print unsorted array
    cout << "Unsorted array: ";
    printArray(arr, n);

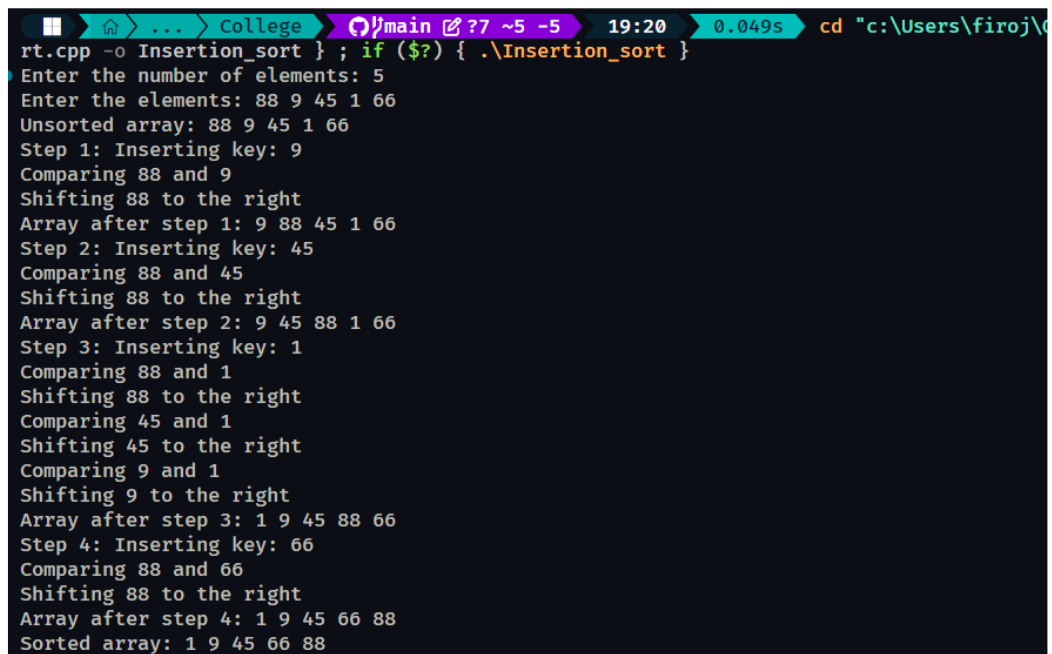
    // Perform Insertion Sort
    insertionSort(arr, n);

    // Print sorted array
    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}

```

7. Output:



```

C:\Users\firoj> g++ rt.cpp -o Insertion_sort } ; if ($?) { .\Insertion_sort }
Enter the number of elements: 5
Enter the elements: 88 9 45 1 66
Unsorted array: 88 9 45 1 66
Step 1: Inserting key: 9
Comparing 88 and 9
Shifting 88 to the right
Array after step 1: 9 88 45 1 66
Step 2: Inserting key: 45
Comparing 88 and 45
Shifting 88 to the right
Array after step 2: 9 45 88 1 66
Step 3: Inserting key: 1
Comparing 88 and 1
Shifting 88 to the right
Comparing 45 and 1
Shifting 45 to the right
Comparing 9 and 1
Shifting 9 to the right
Array after step 3: 1 9 45 88 66
Step 4: Inserting key: 66
Comparing 88 and 66
Shifting 88 to the right
Array after step 4: 1 9 45 66 88
Sorted array: 1 9 45 66 88

```

Lab Report 9: Design and Analysis of Merge Sort Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Merge Sort is an efficient, comparison-based sorting algorithm that uses a divide-and-conquer approach. It divides the array into two halves, recursively sorts each half, and then merges the sorted halves back together. Merge Sort is stable (preserves the relative order of equal elements) and guarantees consistent performance regardless of the input order, making it suitable for large datasets.

2. Key Concepts:

- The algorithm splits the array until sub-arrays are of size 1 (trivially sorted).
- Merging combines two sorted sub-arrays into a single sorted array by comparing elements.
- It requires additional space for temporary arrays during merging, unlike in-place algorithms like Quick Sort.

3. Properties of Merge Sort:

- Performance is consistent across best, worst, and average cases due to balanced splitting.
- It is not an in-place algorithm, requiring $O(n)$ extra space for merging.
- It is widely used in external sorting (e.g., sorting data too large to fit in memory).

4. Algorithm:

Algorithm: Merge_Sort (arr, left, right)

Input: An array arr, and indices left and right defining the sub-array to sort

Output: The array arr sorted in ascending order

1. If left < right:

- Compute $mid = (left + right) / 2$.
 - Call Merge_Sort(arr, left, mid) to sort the left half.
 - Call Merge_Sort(arr, mid+1, right) to sort the right half.
 - Call Merge(arr, left, mid, right) to merge the sorted halves.
-

2. Return the sorted array.

Algorithm: Merge (arr, left, mid, right)

Input: An array arr, and indices left, mid, and right defining the sub-arrays

Output: The merged sub-array in sorted order

1. Create temporary arrays L (left to mid) and R (mid+1 to right).
 2. Set $i = 0$ (index for L), $j = 0$ (index for R), $k = \text{left}$ (index for arr).
 3. While $i < \text{size of L}$ and $j < \text{size of R}$:
 - If $L[i] \leq R[j]$:
 - Set $\text{arr}[k] = L[i]$ and increment i .
 - Else:
 - Set $\text{arr}[k] = R[j]$ and increment j .
 - Increment k .
 4. Copy remaining elements of L (if any) to arr.
 5. Copy remaining elements of R (if any) to arr.
-

5. Time and Space Complexity Analysis:

- **Time Complexity:**
 - **Best Case:** $O(n \log n)$ – Array is split into halves and merged consistently.
 - **Worst Case:** $O(n \log n)$ – Same as best case, due to guaranteed balanced splits.
 - **Average Case:** $O(n \log n)$ – Consistent performance across all inputs.
- **Space Complexity:** $O(n)$ – Extra space is needed for temporary arrays during merging.

6. Code Implementation:

```
/*Program that performs Merge Sort on an array with detailed step-by-
step output*/
#include<iostream>
using namespace std;

// Function to merge two sub-arrays and show the process
void merge(int arr[], int left, int mid, int right, int& step) {
    int n1 = mid - left + 1; // Size of left sub-array
    int n2 = right - mid;     // Size of right sub-array
    int L[n1], R[n2];        // Temporary arrays

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    cout << "Step " << step << ": Merging sub-arrays: ";
    for (int i = 0; i < n1; i++) cout << L[i] << " ";
```

```

    cout << "and ";
    for (int j = 0; j < n2; j++) cout << R[j] << " ";
    cout << endl;

    int i = 0, j = 0, k = left; // Indices for L, R, and main array
    while (i < n1 && j < n2) {
        cout << "Comparing " << L[i] << " and " << R[j] << endl;
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            cout << "Adding " << L[i] << " from left sub-array" <<
endl;
            i++;
        } else {
            arr[k] = R[j];
            cout << "Adding " << R[j] << " from right sub-array" <<
endl;
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[] if any
    while (i < n1) {
        arr[k] = L[i];
        cout << "Adding remaining " << L[i] << " from left sub-array"
<< endl;
        i++;
        k++;
    }

    // Copy remaining elements of R[] if any
    while (j < n2) {
        arr[k] = R[j];
        cout << "Adding remaining " << R[j] << " from right sub-array"
<< endl;
        j++;
        k++;
    }

    // Print array after merging
    cout << "Array after step " << step << ": ";
    for (int x = left; x <= right; x++)
        cout << arr[x] << " ";
    cout << endl;
    step++; // Increment step counter
}

// Function to perform Merge Sort and show the process
void mergeSort(int arr[], int left, int right, int& step) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Avoid overflow
        mergeSort(arr, left, mid, step);

```

```

        mergeSort(arr, mid + 1, right, step);
        merge(arr, left, mid, right, step);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Print unsorted array
    cout << "Unsorted array: ";
    printArray(arr, n);

    // Perform Merge Sort
    int step = 1; // Initialize step counter
    mergeSort(arr, 0, n-1, step);

    // Print sorted array
    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}

```

7. Output:

```
pp -o Merge_sort } ; if ($?) { .\Merge_sort }
Enter the number of elements: 5
Enter the elements: 44 7 4 88 12
Unsorted array: 44 7 4 88 12
Step 1: Merging sub-arrays: 44 and 7
Comparing 44 and 7
Adding 7 from right sub-array
Adding remaining 44 from left sub-array
Array after step 1: 7 44
Step 2: Merging sub-arrays: 7 44 and 4
Comparing 7 and 4
Adding 4 from right sub-array
Adding remaining 7 from left sub-array
Adding remaining 44 from left sub-array
Array after step 2: 4 7 44
Step 3: Merging sub-arrays: 88 and 12
Comparing 88 and 12
Adding 12 from right sub-array
Adding remaining 88 from left sub-array
Array after step 3: 12 88
Step 4: Merging sub-arrays: 4 7 44 and 12 88
Comparing 4 and 12
Adding 4 from left sub-array
Comparing 7 and 12
Adding 7 from left sub-array
Comparing 44 and 12
Adding 12 from right sub-array
Comparing 44 and 88
Adding 44 from left sub-array
Adding remaining 88 from right sub-array
Array after step 4: 4 7 12 44 88
Sorted array: 4 7 12 44 88
```

Lab Report 10: Design and Analysis of Min-Max Sort Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Min-Max Sort is a comparison-based sorting algorithm that improves upon Selection Sort by finding both the minimum and maximum elements in each pass through the unsorted portion of the array. It places the minimum at the left end and the maximum at the right end, reducing the unsorted portion from both sides simultaneously. This bidirectional approach can reduce the number of passes compared to standard Selection Sort, though it still involves extensive comparisons.

2. Key Concepts:

- Min-Max Sort works by identifying the smallest and largest elements in each iteration.
- It swaps these elements to their correct positions at the boundaries of the unsorted portion.
- Like Selection Sort, it is in-place but not stable, as it may alter the relative order of equal elements.

3. Properties of Min-Max Sort:

- Best case occurs when the array is already sorted, minimizing swaps.
- Worst case occurs with a random or reverse-sorted array, requiring full comparisons and swaps.
- It reduces the number of passes to roughly half that of Selection Sort by sorting from both ends.

4. Algorithm:

Algorithm: MinMax Sort (arr, n)

Input: An array arr of size n

Output: The array arr sorted in ascending order

1. For $i = 0$ to $n/2$:

- Set $\text{min_idx} = i$ and $\text{max_idx} = i$ (initialize minimum and maximum indices).
-

- For $j = i$ to $n-1-i$:
-

- If $\text{arr}[j] < \text{arr}[\text{min_idx}]$:

- Set $\text{min_idx} = j$.

- If $\text{arr}[j] > \text{arr}[\text{max_idx}]$:

- Set $\text{max_idx} = j$.

- If $\text{min_idx} \neq i$:
-

- Swap $\text{arr}[i]$ and $\text{arr}[\text{min_idx}]$.

- If $\text{max_idx} = i$ (after min swap):
-

- Update max_idx to point to the new maximum.

- If $\text{max_idx} \neq n-1-i$:
-

- Swap $\text{arr}[n-1-i]$ and $\text{arr}[\text{max_idx}]$.

2. Return the sorted array.

5. Time and Space Complexity Analysis:

- Time Complexity:

- **Best Case:** $O(n^2)$ – Comparisons are still performed, though swaps may be minimized.
- **Worst Case:** $O(n^2)$ – Full comparisons and swaps for random or reverse-sorted input.
- **Average Case:** $O(n^2)$ – Roughly $n^2/4$ comparisons, slightly better than Selection Sort due to dual sorting.

- **Space Complexity:** $O(1)$ – Only a constant amount of extra space is used (e.g., for indices and temporary swaps).

6. Code Implementation:

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

// Function to perform Min-Max Sort and show the process
void minMaxSort(int arr[], int n) {
    int step = 1; // Initialize step counter
    // Traverse through half the array since we sort from both ends
    for (int i = 0; i < n/2; i++) {
        int min_idx = i, max_idx = i;
        cout << "Step " << step << ": Initial minimum: " <<
arr[min_idx] << ", Initial maximum: " << arr[max_idx] << endl;

        // Find min and max in the unsorted portion
        for (int j = i; j < n-i; j++) {
            cout << "Comparing " << arr[j] << " with current min " <<
arr[min_idx] << endl;
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
                cout << "New minimum found: " << arr[min_idx] << endl;
            }
            cout << "Comparing " << arr[j] << " with current max " <<
arr[max_idx] << endl;
            if (arr[j] > arr[max_idx]) {
                max_idx = j;
                cout << "New maximum found: " << arr[max_idx] << endl;
            }
        }

        // Swap minimum to the left end
        if (min_idx != i) {
            int temp = arr[i];
            arr[i] = arr[min_idx];
            arr[min_idx] = temp;
        }
    }
}
```

```

        cout << "Swapped " << arr[i] << " to position " << i <<
endl;
    }

    // Adjust max_idx if it was affected by the min swap
    if (max_idx == i) max_idx = min_idx;

    // Swap maximum to the right end
    if (max_idx != n-1-i) {
        int temp = arr[n-1-i];
        arr[n-1-i] = arr[max_idx];
        arr[max_idx] = temp;
        cout << "Swapped " << arr[n-1-i] << " to position " << n-
1-i << endl;
    }

    // Print array after each pass
    cout << "Array after step " << step << ": ";
    for (int k = 0; k < n; k++)
        cout << arr[k] << " ";
    cout << endl;
    step++; // Increment step counter
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Print unsorted array
    cout << "Unsorted array: ";
    printArray(arr, n);

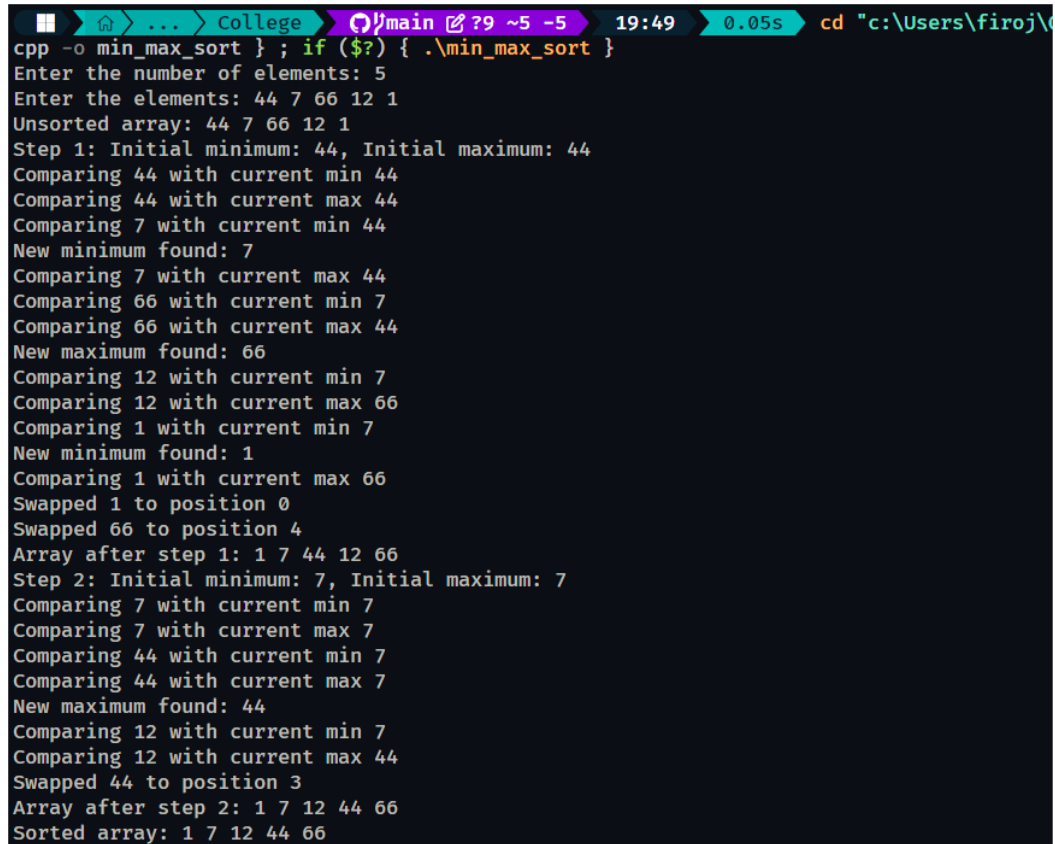
    // Perform Min-Max Sort
    minMaxSort(arr, n);

    // Print sorted array
    cout << "Sorted array: ";

```

```
printArray(arr, n);  
  
return 0;  
}
```

7. Output:



The screenshot shows a terminal window with a dark background. The title bar indicates the file is 'main' and the current directory is 'c:\Users\firoj\'. The program is compiled with 'g++ -o min_max_sort'. The user enters '5' for the number of elements and '44 7 66 12 1' for the elements. The program then displays the unsorted array and proceeds with two steps of the min-max sort algorithm. Step 1 finds the minimum (1) and maximum (66) and swaps them. Step 2 finds the minimum (7) and maximum (44) and swaps them. The final sorted array is '1 7 12 44 66'.

```
cpp -o min_max_sort } ; if ($?) { .\min_max_sort }  
Enter the number of elements: 5  
Enter the elements: 44 7 66 12 1  
Unsorted array: 44 7 66 12 1  
Step 1: Initial minimum: 44, Initial maximum: 44  
Comparing 44 with current min 44  
Comparing 44 with current max 44  
Comparing 7 with current min 44  
New minimum found: 7  
Comparing 7 with current max 44  
Comparing 66 with current min 7  
Comparing 66 with current max 44  
New maximum found: 66  
Comparing 12 with current min 7  
Comparing 12 with current max 66  
Comparing 1 with current min 7  
New minimum found: 1  
Comparing 1 with current max 66  
Swapped 1 to position 0  
Swapped 66 to position 4  
Array after step 1: 1 7 44 12 66  
Step 2: Initial minimum: 7, Initial maximum: 7  
Comparing 7 with current min 7  
Comparing 7 with current max 7  
Comparing 44 with current min 7  
Comparing 44 with current max 7  
New maximum found: 44  
Comparing 12 with current min 7  
Comparing 12 with current max 44  
Swapped 44 to position 3  
Array after step 2: 1 7 12 44 66  
Sorted array: 1 7 12 44 66
```


Lab Report 11: Design and Analysis of Binary Search Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

Binary Search is an efficient algorithm for finding a target element in a sorted array. It works by repeatedly dividing the search range in half, comparing the target with the middle element, and then narrowing the search to the left or right half based on the comparison. Binary Search is significantly faster than Linear Search for large datasets but requires the array to be sorted beforehand.

2. Key Concepts:

- Binary Search relies on the array being sorted in ascending order.
- It eliminates half of the remaining elements with each comparison, making it logarithmic in complexity.
- It can be implemented iteratively or recursively; the iterative version is often preferred for simplicity and space efficiency.

3. Properties of Binary Search:

- Best case occurs when the target is the middle element of the initial range.
- Worst case occurs when the target is at the edges or not present, requiring maximum divisions.
- It is not suitable for unsorted arrays or dynamic data without preprocessing.

4. Algorithm:

Algorithm: Binary_Search (arr, n, target)

Input: A sorted array arr of size n, and a target value target

Output: The index of target in arr, or -1 if not found

1. Set left = 0 and right = n-1 (initial search range).

2. While left ≤ right:

- Compute mid = (left + right) / 2.

- If arr[mid] = target:

 - Return mid.

- If arr[mid] > target:

 - Set right = mid - 1 (search left half).

- Else:

 - Set left = mid + 1 (search right half).

3. Return -1 (target not found).

5. Time and Space Complexity Analysis:

- Time Complexity:

- **Best Case:** $O(1)$ – Target is found at the first middle element.
- **Worst Case:** $O(\log n)$ – Search range halves until the target is found or exhausted.
- **Average Case:** $O(\log n)$ – Roughly $\log_2(n)$ comparisons for a random target.

- **Space Complexity:** $O(1)$ – Only a constant amount of extra space is used (e.g., for indices).

6. Code Implementation:

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

// Function to perform Binary Search and show the process
int binarySearch(int arr[], int n, int target) {
    int step = 1; // Initialize step counter
    int left = 0, right = n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2; // Avoid overflow
        cout << "Step " << step << ": Search range [" << left << ", "
<< right << "], Mid: " << mid
        << ", Value at mid: " << arr[mid] << endl;

        // Print comparison
        cout << "Comparing " << arr[mid] << " with target " << target
<< endl;

        if (arr[mid] == target) {
            cout << "Target " << target << " found at index " << mid
<< endl;
            return mid;
        }
        else if (arr[mid] > target) {
            cout << "Value " << arr[mid] << " is greater than target,
searching left half" << endl;
            right = mid - 1;
        }
        else {
            cout << "Value " << arr[mid] << " is less than target,
searching right half" << endl;
            left = mid + 1;
        }

        // Print array with current search range
        cout << "Array after step " << step << ": ";
```

```

        for (int i = 0; i < n; i++) {
            if (i >= left && i <= right)
                cout << arr[i] << " "; // Highlight current range
            else
                cout << "_ ";          // Indicate out-of-range
elements
        }
        cout << endl;
        step++; // Increment step counter
    }

    cout << "Target " << target << " not found in the array" << endl;
    return -1;
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int n, target;
    cout << "Enter the number of elements (array must be sorted): ";
    cin >> n;

    int arr[n];
    cout << "Enter the sorted elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Print the array
    cout << "Sorted array: ";
    printArray(arr, n);

    cout << "Enter the target value to search: ";
    cin >> target;

    // Perform Binary Search
    int result = binarySearch(arr, n, target);

    if (result != -1)
        cout << "Final result: Target found at index " << result <<
endl;
    else
        cout << "Final result: Target not found" << endl;

    return 0;
}

```

7. Output:

```
cpp -o binary_search } ; if ($?) { .\binary_search }
Enter the number of elements (array must be sorted): 5
Enter the sorted elements: 1 8 12 22 50
Sorted array: 1 8 12 22 50
Enter the target value to search: 22
Step 1: Search range [0, 4], Mid: 2, Value at mid: 12
Comparing 12 with target 22
Value 12 is less than target, searching right half
Array after step 1: _ _ _ 22 50
Step 2: Search range [3, 4], Mid: 3, Value at mid: 22
Comparing 22 with target 22
Target 22 found at index 3
Final result: Target found at index 3
```

Lab Report 12: Design and Analysis of Fractional Knapsack Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

The Fractional Knapsack problem involves maximizing the total value of items placed in a knapsack of limited capacity, where fractions of items can be taken. It is solved using a greedy approach: items are sorted by their value-to-weight ratio in descending order, and the knapsack is filled by taking as much as possible of the highest-ratio items first. This contrasts with the 0/1 Knapsack problem, which requires dynamic programming due to its discrete nature.

2. Key Concepts:

- The algorithm prioritizes items based on their value per unit weight (value/weight ratio).
- It allows partial inclusion of items, making it greedy and optimal for this specific problem.
- Sorting by ratio ensures the maximum possible value is achieved within the capacity constraint.

3. Properties of Fractional Knapsack:

- Best case occurs when all items can be fully included without exceeding capacity.
- Worst case involves taking fractions of multiple items, but the greedy choice remains optimal.
- It assumes items can be divided continuously (e.g., like grains or liquids), unlike discrete objects.

4. Algorithm:

Algorithm: Fractional Knapsack (items, n, capacity)

Input: An array items of size n (each with value and weight), and knapsack capacity capacity

Output: The maximum value achievable

1. For each item in items:

- Compute $\text{ratio} = \text{value} / \text{weight}$.

2. Sort items by ratio in descending order.

3. Set $\text{current_weight} = 0$ and $\text{total_value} = 0$.

4. For $i = 0$ to $n - 1$:

- If $\text{current_weight} + \text{items}[i].\text{weight} \leq \text{capacity}$:

- Add $\text{items}[i].\text{value}$ to total_value .
- Add $\text{items}[i].\text{weight}$ to current_weight .

- Else:

- Compute $\text{fraction} = (\text{capacity} - \text{current_weight}) / \text{items}[i].\text{weight}$.
- Add $\text{fraction} * \text{items}[i].\text{value}$ to total_value .
- Break (knapsack is full).

5. Return total_value .

5. Time and Space Complexity Analysis:

- **Time Complexity:**
 - $O(n \log n)$ – Dominated by the sorting step; the greedy selection is $O(n)$.
 - **Best/Worst/Average Cases** are all $O(n \log n)$ due to consistent sorting overhead.
- **Space Complexity:** $O(1)$ – Only a constant amount of extra space is used (excluding input storage), assuming in-place sorting.

6. Code Implementation:

```
#include<iostream>
#include<algorithm> // For sort
using namespace std;

// Structure to represent an item
struct Item {
    int value, weight;
    double ratio; // Value-to-weight ratio
};

// Function to compare items by ratio for sorting
bool compare(Item a, Item b) {
    return a.ratio > b.ratio; // Descending order
}

// Function to solve Fractional Knapsack and show the process
double fractionalKnapsack(Item arr[], int n, int capacity) {
    int step = 1; // Initialize step counter
    double total_value = 0.0;
    int current_weight = 0;

    // Step 1: Compute value-to-weight ratios
    cout << "Step " << step << ": Computing value-to-weight ratios" <<
endl;
    for (int i = 0; i < n; i++) {
        arr[i].ratio = (double)arr[i].value / arr[i].weight;
        cout << "Item " << i << ": Value = " << arr[i].value << ",
Weight = " << arr[i].weight
        << ", Ratio = " << arr[i].ratio << endl;
    }
    step++;

    // Step 2: Sort items by ratio
    cout << "Step " << step << ": Sorting items by ratio (descending)"
<< endl;
    sort(arr, arr + n, compare);
    for (int i = 0; i < n; i++) {
```

```

        cout << "Item " << i << ": Value = " << arr[i].value << ",
Weight = " << arr[i].weight
        << ", Ratio = " << arr[i].ratio << endl;
    }
    step++;

    // Step 3: Fill the knapsack
    cout << "Step " << step << ": Filling knapsack (capacity = " <<
capacity << ")" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Considering item " << i << ": Value = " <<
arr[i].value << ", Weight = " << arr[i].weight
        << ", Ratio = " << arr[i].ratio << endl;
        cout << "Current weight = " << current_weight << ", Remaining
capacity = " << capacity - current_weight << endl;

        if (current_weight + arr[i].weight <= capacity) {
            current_weight += arr[i].weight;
            total_value += arr[i].value;
            cout << "Fully adding item: Value = " << arr[i].value <<
", Weight = " << arr[i].weight << endl;
        } else {
            double fraction = (double)(capacity - current_weight) /
arr[i].weight;
            total_value += arr[i].value * fraction;
            current_weight = capacity; // Knapsack is full
            cout << "Adding fraction " << fraction << " of item: Value
= " << arr[i].value * fraction
            << ", Weight = " << capacity - current_weight +
arr[i].weight * fraction << endl;
            break; // No more capacity
        }

        cout << "Knapsack after step " << step << ": Total Value = "
<< total_value
        << ", Total Weight = " << current_weight << endl;
        step++;
    }

    return total_value;
}

int main() {
    int n, capacity;
    cout << "Enter the number of items: ";
    cin >> n;

    Item arr[n];
    cout << "Enter value and weight for each item:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Item " << i << " - Value: ";
        cin >> arr[i].value;
    }
}

```

```

        cout << "Item " << i << " - Weight: ";
        cin >> arr[i].weight;
    }

    cout << "Enter the knapsack capacity: ";
    cin >> capacity;

    // Perform Fractional Knapsack
    double max_value = fractionalKnapsack(arr, n, capacity);

    // Print final result
    cout << "Maximum value achievable: " << max_value << endl;

    return 0;
}

```

7. Output:

```

o Knapsack } ; if ($?) { .\Knapsack }
Enter the number of items: 5
Enter value and weight for each item:
Item 0 - Value: 4
Item 0 - Weight: 1
Item 1 - Value: 5
Item 1 - Weight: 2
Item 2 - Value: 9
Item 2 - Weight: 3
Item 3 - Value: 8
Item 3 - Weight: 1
Item 4 - Value: 15
Item 4 - Weight: 6
Enter the knapsack capacity: 3
Step 1: Computing value-to-weight ratios
Item 0: Value = 4, Weight = 1, Ratio = 4
Item 1: Value = 5, Weight = 2, Ratio = 2.5
Item 2: Value = 9, Weight = 3, Ratio = 3
Item 3: Value = 8, Weight = 1, Ratio = 8
Item 4: Value = 15, Weight = 6, Ratio = 2.5
Step 2: Sorting items by ratio (descending)
Item 0: Value = 8, Weight = 1, Ratio = 8
Item 1: Value = 4, Weight = 1, Ratio = 4
Item 2: Value = 9, Weight = 3, Ratio = 3
Item 3: Value = 5, Weight = 2, Ratio = 2.5
Item 4: Value = 15, Weight = 6, Ratio = 2.5
Step 3: Filling knapsack (capacity = 3)
Considering item 0: Value = 8, Weight = 1, Ratio = 8
Current weight = 0, Remaining capacity = 3
Fully adding item: Value = 8, Weight = 1
Knapsack after step 3: Total Value = 8, Total Weight = 1
Considering item 1: Value = 4, Weight = 1, Ratio = 4
Current weight = 1, Remaining capacity = 2
Fully adding item: Value = 4, Weight = 1
Knapsack after step 4: Total Value = 12, Total Weight = 2
Considering item 2: Value = 9, Weight = 3, Ratio = 3
Current weight = 2, Remaining capacity = 1
Adding fraction 0.333333 of item: Value = 3, Weight = 1
Maximum value achievable: 15

```


Lab Report 13: Design and Analysis of Job Sequencing with Deadline Algorithm

Date: April 01, 2025

Day: Tuesday

1. Theory:

The Job Sequencing with Deadlines problem involves scheduling a set of jobs, each with a profit and a deadline, to maximize total profit. The algorithm assumes each job takes one unit of time, and a job must be completed by its deadline to earn its profit. A greedy approach is used: jobs are sorted by profit in descending order, and each job is scheduled in the latest possible time slot before its deadline, ensuring maximum utilization of available time.

2. Key Concepts:

- Jobs are prioritized based on profit, favoring higher-profit jobs first.
- Time slots are filled from the latest possible position backward to accommodate as many jobs as possible.
- The algorithm assumes deadlines are given as integers representing the maximum time units allowed (e.g., deadline 2 means the job must be completed by time 2).

3. Properties of Job Sequencing with Deadlines:

- Best case occurs when all jobs can be scheduled within their deadlines.
- Worst case occurs when deadlines are tight, limiting the number of schedulable jobs.
- The greedy choice of highest profit ensures an optimal solution for this problem.

4. Algorithm:

Algorithm: Job Sequencing (jobs, n)

Input: An array jobs of size n (each with profit and deadline)

Output: A sequence of jobs maximizing total profit

1. Sort jobs by profit in descending order.
 2. Find max_deadline (the maximum deadline among all jobs).
 3. Initialize an array slots of size max_deadline with -1 (empty slots).
 4. Set total_profit = 0.
 5. For i = 0 to n - 1:
 - For j = min(max_deadline, jobs[i].deadline) - 1 down to 0:
 - If slots[j] = -1 (slot is free):
 - Set slots[j] = i (assign job to slot).
 - Add jobs[i].profit to total_profit.
 - Break (job scheduled).
 6. Return slots and total_profit.
-

5. Time and Space Complexity Analysis:

- Time Complexity:

- $O(n \log n + n * d)$ – Sorting takes $O(n \log n)$, and scheduling takes $O(n * d)$ where d is the maximum deadline.
- **Best/Worst/Average Cases** depend on d , but sorting dominates unless d is very large.

- Space Complexity: $O(d)$ – Extra space for the slots array, where d is the maximum deadline.

6. Code Implementation:

```
#include<iostream>
#include<algorithm> // For sort
using namespace std;

// Structure to represent a job
struct Job {
    int id, profit, deadline;
};

// Function to compare jobs by profit for sorting
bool compare(Job a, Job b) {
    return a.profit > b.profit; // Descending order
}

// Function to solve Job Sequencing and show the process
void jobSequencing(Job arr[], int n) {
    int step = 1; // Initialize step counter
    int total_profit = 0;

    // Step 1: Sort jobs by profit
    cout << "Step " << step << ": Sorting jobs by profit (descending)"
    << endl;
    sort(arr, arr + n, compare);
    for (int i = 0; i < n; i++) {
        cout << "Job " << arr[i].id << ": Profit = " << arr[i].profit
        << ", Deadline = " << arr[i].deadline << endl;
    }
    step++;

    // Step 2: Find maximum deadline
    int max_deadline = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i].deadline > max_deadline)
            max_deadline = arr[i].deadline;
    }
    cout << "Step " << step << ": Maximum deadline = " << max_deadline
    << endl;
    step++;
}
```

```

// Step 3: Initialize slots
int slots[max_deadline];
for (int i = 0; i < max_deadline; i++)
    slots[i] = -1; // -1 indicates empty slot

// Step 4: Schedule jobs
cout << "Step " << step << ": Scheduling jobs" << endl;
for (int i = 0; i < n; i++) {
    cout << "Considering Job " << arr[i].id << ": Profit = " <<
arr[i].profit
    << ", Deadline = " << arr[i].deadline << endl;

    // Try to schedule job in the latest possible slot
    for (int j = min(max_deadline, arr[i].deadline) - 1; j >= 0;
j--) {
        cout << "Checking slot " << j << ": ";
        if (slots[j] == -1) {
            slots[j] = arr[i].id;
            total_profit += arr[i].profit;
            cout << "Assigned to Job " << arr[i].id << endl;

            // Print current schedule
            cout << "Schedule after step " << step << ": ";
            for (int k = 0; k < max_deadline; k++) {
                if (slots[k] == -1) cout << "_ ";
                else cout << "J" << slots[k] << " ";
            }
            cout << "(Profit = " << total_profit << ")" << endl;
            step++;
            break;
        } else {
            cout << "Occupied by Job " << slots[j] << endl;
        }
    }
}

// Print final schedule
cout << "Final Schedule: ";
for (int i = 0; i < max_deadline; i++) {
    if (slots[i] != -1)
        cout << "J" << slots[i] << " ";
}
cout << endl << "Maximum profit achievable: " << total_profit <<
endl;
}

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

    Job arr[n];

```

```
    cout << "Enter job ID, profit, and deadline for each job:" <<
endl;
    for (int i = 0; i < n; i++) {
        cout << "Job " << i << " - ID: ";
        cin >> arr[i].id;
        cout << "Job " << i << " - Profit: ";
        cin >> arr[i].profit;
        cout << "Job " << i << " - Deadline: ";
        cin >> arr[i].deadline;
    }

    // Perform Job Sequencing
    jobSequencing(arr, n);

    return 0;
}
```

7. Output:

```
main ?11 ~5 -5 20:17 44.805s cd "c:\Users\firoj\
eadline.cpp -o job_seq_with_deadline } ; if ($?) { .\job_seq_with_deadline }
Enter the number of jobs: 5
Enter job ID, profit, and deadline for each job:
Job 0 - ID: 2
Job 0 - Profit: 14
Job 0 - Deadline: 3
Job 1 - ID: 0
Job 1 - Profit: 100
Job 1 - Deadline: 1
Job 2 - ID: 1
Job 2 - Profit: 50
Job 2 - Deadline: 2
Job 3 - ID: 3
Job 3 - Profit: 14
Job 3 - Deadline: 2
Job 4 - ID: 4
Job 4 - Profit: 56
Job 4 - Deadline: 2
Step 1: Sorting jobs by profit (descending)
Job 0: Profit = 100, Deadline = 1
Job 4: Profit = 56, Deadline = 2
Job 1: Profit = 50, Deadline = 2
Job 2: Profit = 14, Deadline = 3
Job 3: Profit = 14, Deadline = 2
Step 2: Maximum deadline = 3
Step 3: Scheduling jobs
Considering Job 0: Profit = 100, Deadline = 1
Checking slot 0: Assigned to Job 0
Schedule after step 3: J0 _ _ (Profit = 100)
Considering Job 4: Profit = 56, Deadline = 2
Checking slot 1: Assigned to Job 4
Schedule after step 4: J0 J4 _ (Profit = 156)
Considering Job 1: Profit = 50, Deadline = 2
Checking slot 1: Occupied by Job 4
Checking slot 0: Occupied by Job 0
Considering Job 2: Profit = 14, Deadline = 3
Checking slot 2: Assigned to Job 2
Schedule after step 5: J0 J4 J2 (Profit = 170)
Considering Job 3: Profit = 14, Deadline = 2
Checking slot 1: Occupied by Job 4
Checking slot 0: Occupied by Job 0
Final Schedule: J0 J4 J2
Maximum profit achievable: 170
```

Lab Report 14: Design and Analysis of Heap Sort Algorithm

Date: April 06, 2025

Day: Sunday

1. Theory:

Heap Sort is an efficient sorting algorithm that leverages a binary max-heap (or min-heap) to sort an array. It first builds a max-heap from the array, where the largest element is at the root. Then, it repeatedly extracts the maximum element (root), places it at the end of the array, and reduces the heap size, restoring the heap property each time. This process sorts the array in ascending order. Heap Sort is in-place and has consistent performance, making it reliable for various input sizes.

2. Key Concepts:

- A max-heap ensures the parent node is greater than or equal to its children.
- The algorithm has two phases: building the heap and extracting elements to sort.
- It is not stable (may change the relative order of equal elements) but is in-place, requiring no extra array storage beyond the heap structure.

3. Properties of Heap Sort:

- Best, worst, and average cases have the same time complexity due to the heap-building and extraction process.
- It performs well for large datasets but is slower than Quick Sort in practice due to poor cache locality.
- The heapify operation ensures the heap property is maintained after each swap.

4. Algorithm:

Algorithm: Heap_Sort (arr, n)

Input: An array arr of size n

Output: The array arr sorted in ascending order

1. For $i = n/2 - 1$ down to 0:

 ○ Call Heapify(arr, n, i) to build a max-heap.

2. For $i = n - 1$ down to 1:

 ○ Swap arr[0] (maximum) with arr[i] (move to sorted portion).

 ○ Call Heapify(arr, i, 0) to restore heap property on reduced heap.

3. Return the sorted array.

Algorithm: Heapify (arr, n, i)

Input: An array arr, size n , and index i to heapify

Output: A max-heap rooted at i

1. Set $\text{largest} = i$ (root of subtree).
 2. Set $\text{left} = 2i + 1$ and $\text{right} = 2i + 2$ (children indices).
 3. If $\text{left} < n$ and $\text{arr}[\text{left}] > \text{arr}[\text{largest}]$:
 - o Set $\text{largest} = \text{left}$.
 4. If $\text{right} < n$ and $\text{arr}[\text{right}] > \text{arr}[\text{largest}]$:
 - o Set $\text{largest} = \text{right}$.
 5. If $\text{largest} \neq i$:
 - o Swap $\text{arr}[i]$ and $\text{arr}[\text{largest}]$.
 - o Call $\text{Heapify}(\text{arr}, n, \text{largest})$ recursively.
-

5. Time and Space Complexity Analysis:

- Time Complexity:

- **Best Case:** $O(n \log n)$ – Building the heap is $O(n)$, and n extractions are $O(\log n)$ each.
- **Worst Case:** $O(n \log n)$ – Same as best case, as heap operations are consistent.
- **Average Case:** $O(n \log n)$ – Uniform performance across inputs.

- **Space Complexity:** $O(1)$ – In-place sorting, with only recursive stack space ($O(\log n)$ in the worst case, but iterative versions can reduce this to $O(1)$).

6. Code Implementation:

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

// Function to heapify a subtree rooted at index i
void heapify(int arr[], int n, int i, int& step) {
    int largest = i;      // Initialize largest as root
    int left = 2 * i + 1;  // Left child
    int right = 2 * i + 2; // Right child

    cout << "Step " << step << ": Heapifying at index " << i << ", Value = " << arr[i] << endl;

    // Compare with left child
    if (left < n) {
        cout << "Comparing " << arr[i] << " with left child " << arr[left] << endl;
        if (arr[left] > arr[largest]) {
            largest = left;
            cout << "Left child " << arr[left] << " is larger" << endl;
        }
    }

    // Compare with right child
    if (right < n) {
        cout << "Comparing " << arr[largest] << " with right child " << arr[right] << endl;
        if (arr[right] > arr[largest]) {
            largest = right;
            cout << "Right child " << arr[right] << " is larger" << endl;
        }
    }
}
```



```

    }

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        cout << "Swapped " << arr[i] << " with " << arr[largest] << endl;

        // Print array after swap
        cout << "Array after step " << step << ": ";
        for (int k = 0; k < n; k++)
            cout << arr[k] << " ";
        cout << endl;
        step++;

        // Recursively heapify the affected subtree
        heapify(arr, n, largest, step);
    } else {
        cout << "No swap needed" << endl;
        cout << "Array after step " << step << ": ";
        for (int k = 0; k < n; k++)
            cout << arr[k] << " ";
        cout << endl;
        step++;
    }
}

// Function to perform Heap Sort and show the process
void heapSort(int arr[], int n) {
    int step = 1; // Initialize step counter

    // Step 1: Build max-heap
    cout << "Building max-heap:" << endl;
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i, step);
    }

    // Step 2: Extract elements from heap
    cout << "Extracting elements to sort:" << endl;
    for (int i = n - 1; i > 0; i--) {
        cout << "Step " << step << ": Moving max " << arr[0] << " to position " << i <<
endl;
        swap(arr[0], arr[i]);

        // Print array after swap
        cout << "Array after step " << step << ": ";
        for (int k = 0; k < n; k++)
            cout << arr[k] << " ";
        cout << endl;
        step++;

        // Heapify the reduced heap
        heapify(arr, i, 0, step);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

```

```

int arr[n];
cout << "Enter the elements: ";
for (int i = 0; i < n; i++) {
    cin >> arr[i];
}

// Print unsorted array
cout << "Unsorted array: ";
printArray(arr, n);

// Perform Heap Sort
heapSort(arr, n);

// Print sorted array
cout << "Sorted array: ";
printArray(arr, n);

return 0;
}

```

7. Output:

```

C:\Users\firoj\main ?3 ~6 -5 14:52 0.063s cd "c:\Users\firoj\
p -o heap_sort } ; if ($?) { .\heap_sort }
Enter the number of elements: 3
Enter the elements: 88 5 12
Unsorted array: 88 5 12
Building max-heap:
Step 1: Heapifying at index 0, Value = 88
Comparing 88 with left child 5
Comparing 88 with right child 12
No swap needed
Array after step 1: 88 5 12
Extracting elements to sort:
Step 2: Moving max 88 to position 2
Array after step 2: 12 5 88
Step 3: Heapifying at index 0, Value = 12
Comparing 12 with left child 5
No swap needed
Array after step 3: 12 5
Step 4: Moving max 12 to position 1
Array after step 4: 5 12 88
Step 5: Heapifying at index 0, Value = 5
No swap needed
Array after step 5: 5
Sorted array: 5 12 88

```

Lab Report 15: Design and Analysis of Kruskal's Algorithm

Date: April 06, 2025

Day: Sunday

1. Theory:

Kruskal's Algorithm is a greedy algorithm that finds the Minimum Spanning Tree (MST) of a connected, undirected graph with weighted edges. The MST is a subset of edges that connects all vertices with the minimum total edge weight and no cycles. The algorithm sorts all edges by weight, then iteratively adds the smallest edge to the MST if it doesn't form a cycle, using a disjoint-set data structure to detect cycles efficiently.

2. Key Concepts:

- Edges are processed in non-decreasing order of weight.
- A disjoint-set (union-find) structure tracks connected components to avoid cycles.
- The algorithm ensures the MST spans all vertices with the least possible total weight.

3. Properties of Kruskal's Algorithm:

- It works on connected graphs and produces a unique MST if all edge weights are distinct.
- Best case occurs with already sorted edges or a sparse graph; worst case involves dense graphs.
- It's optimal for finding the MST in terms of total weight.

4. Algorithm:

Algorithm: Kruskal_MST (graph, V, E)

Input: A graph with V vertices and E edges (each with source, destination, weight)

Output: The Minimum Spanning Tree edges and total weight

1. Sort all edges in graph by weight in ascending order.
 2. Initialize a disjoint-set for V vertices (each vertex in its own set).
 3. Set mst_edges = empty and mst_weight = 0.
 4. For each edge (u, v, w) in sorted order:
 - If find(u) \neq find(v) (no cycle formed):
 - Add edge (u, v) to mst_edges.
 - Add w to mst_weight.
 - Union u and v in the disjoint-set.
 5. Return mst_edges and mst_weight.
-

5. Time and Space Complexity Analysis:

- Time Complexity:

- $O(E \log E)$ or $O(E \log V)$ – Sorting edges takes $O(E \log E)$, and union-find operations take $O(\alpha(V))$ per edge, where α is the inverse Ackermann function (*nearly constant*).
- **Best/Worst/Average Cases** are dominated by sorting, so $O(E \log E)$.

- Space Complexity: $O(V + E)$ – Space for the edge list and disjoint-set structure.

6. Code Implementation:

```
#include<iostream>
#include<algorithm> // For sort
using namespace std;

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure for disjoint-set
struct DisjointSet {
    int *parent, *rank;
    int n;

    DisjointSet(int n) {
        this->n = n;
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i; // Each vertex is its own parent
            rank[i] = 0;
        }
    }

    int find(int u) {
        if (parent[u] != u)
            parent[u] = find(parent[u]); // Path compression
        return parent[u];
    }

    void unionSet(int u, int v) {
        int pu = find(u), pv = find(v);
        if (pu == pv) return;
        if (rank[pu] < rank[pv])
            parent[pu] = pv;
        else if (rank[pu] > rank[pv])
            parent[pv] = pu;
        else {
            parent[pv] = pu;
            rank[pu]++;
        }
    }
};

// Function to compare edges by weight
bool compare(Edge a, Edge b) {
    return a.weight < b.weight;
}

// Function to perform Kruskal's Algorithm with concise output
```

```

void kruskalMST(Edge edges[], int V, int E) {
    int step = 1;

    // Step 1: Sort edges by weight
    cout << "Step " << step++ << ": Sorting edges by weight" << endl;
    sort(edges, edges + E, compare);
    for (int i = 0; i < E; i++)
        cout << "Edge " << edges[i].src << " - " << edges[i].dest << ": " <<
edges[i].weight << endl;

    // Step 2: Initialize disjoint-set and MST
    DisjointSet ds(V);
    Edge mst[V - 1]; // MST will have V-1 edges
    int mst_weight = 0, mst_index = 0;

    // Step 3: Process edges
    cout << "Step " << step++ << ": Building MST" << endl;
    for (int i = 0; i < E && mst_index < V - 1; i++) {
        int u = edges[i].src, v = edges[i].dest, w = edges[i].weight;
        cout << "Considering edge " << u << " - " << v << " (weight " << w << "): ";

        if (ds.find(u) != ds.find(v)) {
            mst[mst_index++] = edges[i];
            mst_weight += w;
            ds.unionSet(u, v);
            cout << "Added to MST" << endl;
            cout << "MST after step " << step++ << ": ";
            for (int j = 0; j < mst_index; j++)
                cout << mst[j].src << "-" << mst[j].dest << " ";
            cout << "(Total weight = " << mst_weight << ")" << endl;
        } else {
            cout << "Skipped (forms cycle)" << endl;
        }
    }

    // Print final MST
    cout << "Final MST: ";
    for (int i = 0; i < V - 1; i++)
        cout << mst[i].src << "-" << mst[i].dest << " ";
    cout << endl << "Total weight: " << mst_weight << endl;
}

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    Edge edges[E];
    cout << "Enter " << E << " edges (source destination weight):" << endl;
    for (int i = 0; i < E; i++) {
        cout << "Edge " << i + 1 << ": ";
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }

    cout << "Graph edges:" << endl;
    for (int i = 0; i < E; i++)
        cout << edges[i].src << " - " << edges[i].dest << ": " << edges[i].weight << endl;

    // Perform Kruskal's Algorithm
    kruskalMST(edges, V, E);

    return 0;
}

```

7. Output:

```
kruskals } ; if ($?) { .\kruskals }
Enter the number of vertices: 4
Enter the number of edges: 5
Enter 5 edges (source destination weight):
Edge 1: 0 1 10
Edge 2: 0 2 6
Edge 3: 0 3 5
Edge 4: 1 3 15
Edge 5: 2 3 4
Graph edges:
0 - 1: 10
0 - 2: 6
0 - 3: 5
1 - 3: 15
2 - 3: 4
Step 1: Sorting edges by weight
Edge 2 - 3: 4
Edge 0 - 3: 5
Edge 0 - 2: 6
Edge 0 - 1: 10
Edge 1 - 3: 15
Step 2: Building MST
Considering edge 2 - 3 (weight 4): Added to MST
MST after step 3: 2-3 (Total weight = 4)
Considering edge 0 - 3 (weight 5): Added to MST
MST after step 4: 2-3 0-3 (Total weight = 9)
Considering edge 0 - 2 (weight 6): Skipped (forms cycle)
Considering edge 0 - 1 (weight 10): Added to MST
MST after step 5: 2-3 0-3 0-1 (Total weight = 19)
Final MST: 2-3 0-3 0-1
Total weight: 19
```

Lab Report 16: Design and Analysis of Prim's Algorithm

Date: April 06, 2025

Day: Sunday

1. Theory:

Prim's Algorithm is a greedy algorithm that constructs the Minimum Spanning Tree (MST) of a connected, undirected graph with weighted edges. Starting from an arbitrary vertex, it incrementally adds the edge with the smallest weight that connects a visited vertex to an unvisited one, ensuring no cycles are formed. The result is a tree that spans all vertices with the minimum total edge weight.

2. Key Concepts:

- It grows the MST one edge at a time, always choosing the minimum-weight edge to an unvisited vertex.
- A visited set (or key array) tracks which vertices are included, avoiding cycles implicitly.
- It's particularly efficient for dense graphs when implemented with a priority queue (though we'll use a simple array-based approach here for clarity).

3. Properties of Prim's Algorithm:

- It produces a unique MST if all edge weights are distinct and the graph is connected.
- Best case occurs with a sparse graph or low-degree vertices; worst case involves dense graphs.
- It's optimal for finding the MST and is often used in network optimization (e.g., laying pipelines).

4. Algorithm:

Algorithm: Prim MST (graph, V)

Input: A graph with V vertices represented as an adjacency matrix

Output: The Minimum Spanning Tree edges and total weight

1. Initialize key[V] with infinity (minimum weight to include each vertex), parent[V] with -1 (to store MST edges), and visited[V] with false.
 2. Set key[0] = 0 (start from vertex 0).
 3. For count = 0 to V-1:
 - Find the unvisited vertex u with the minimum key[u].
 - Mark visited[u] = true.
 - For each vertex v adjacent to u:
 - If v is unvisited and graph[u][v] < key[v]:
 - Update key[v] = graph[u][v].
 - Set parent[v] = u.
 4. Return parent array (MST edges) and compute total weight from key.
-

5. Time and Space Complexity Analysis:

- Time Complexity:

- $O(V^2)$ – Using an adjacency matrix and linear search for the minimum key (as implemented here).
- Can be improved to $O(E \log V)$ with a priority queue, but this version keeps it simple.

- Space Complexity: $O(V)$ – Space for key, parent, and visited arrays.

6. Code Implementation:

```
#include<iostream>
#include<algorithm> // For sort
using namespace std;

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure for disjoint-set
struct DisjointSet {
    int *parent, *rank;
    int n;

    DisjointSet(int n) {
        this->n = n;
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i; // Each vertex is its own parent
            rank[i] = 0;
        }
    }

    int find(int u) {
        if (parent[u] != u)
            parent[u] = find(parent[u]); // Path compression
        return parent[u];
    }

    void unionSet(int u, int v) {
        int pu = find(u), pv = find(v);
        if (pu == pv) return;
        if (rank[pu] < rank[pv])
            parent[pu] = pv;
        else if (rank[pu] > rank[pv])
            parent[pv] = pu;
        else {
            parent[pv] = pu;
            rank[pu]++;
        }
    }
};

// Function to compare edges by weight
bool compare(Edge a, Edge b) {
    return a.weight < b.weight;
}

// Function to perform Kruskal's Algorithm with concise output
```



```

void kruskalMST(Edge edges[], int V, int E) {
    int step = 1;

    // Step 1: Sort edges by weight
    cout << "Step " << step++ << ": Sorting edges by weight" << endl;
    sort(edges, edges + E, compare);
    for (int i = 0; i < E; i++)
        cout << "Edge " << edges[i].src << " - " << edges[i].dest << ": " <<
edges[i].weight << endl;

    // Step 2: Initialize disjoint-set and MST
    DisjointSet ds(V);
    Edge mst[V - 1]; // MST will have V-1 edges
    int mst_weight = 0, mst_index = 0;

    // Step 3: Process edges
    cout << "Step " << step++ << ": Building MST" << endl;
    for (int i = 0; i < E && mst_index < V - 1; i++) {
        int u = edges[i].src, v = edges[i].dest, w = edges[i].weight;
        cout << "Considering edge " << u << " - " << v << " (weight " << w << "): ";

        if (ds.find(u) != ds.find(v)) {
            mst[mst_index++] = edges[i];
            mst_weight += w;
            ds.unionSet(u, v);
            cout << "Added to MST" << endl;
            cout << "MST after step " << step++ << ": ";
            for (int j = 0; j < mst_index; j++)
                cout << mst[j].src << "-" << mst[j].dest << " ";
            cout << "(Total weight = " << mst_weight << ")" << endl;
        } else {
            cout << "Skipped (forms cycle)" << endl;
        }
    }

    // Print final MST
    cout << "Final MST: ";
    for (int i = 0; i < V - 1; i++)
        cout << mst[i].src << "-" << mst[i].dest << " ";
    cout << endl << "Total weight: " << mst_weight << endl;
}

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    Edge edges[E];
    cout << "Enter " << E << " edges (source destination weight):" << endl;
    for (int i = 0; i < E; i++) {
        cout << "Edge " << i + 1 << ": ";
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }

    cout << "Graph edges:" << endl;
    for (int i = 0; i < E; i++)
        cout << edges[i].src << " - " << edges[i].dest << ": " << edges[i].weight << endl;

    // Perform Kruskal's Algorithm
    kruskalMST(edges, V, E);

    return 0;
}

```

7. Output:

```
main ?4 ~6 -5 15:03 28.828s cd "c:\Users\firoj\
rims }
Enter the number of vertices: 4
Enter the adjacency matrix (4x4):
Use 99999 for no edge between vertices
0 10 6 5
10 0 99999 15
6 99999 0 4
5 15 4 0
Graph adjacency matrix:
0 10 6 5
10 0 99999 15
6 99999 0 4
5 15 4 0
Step 1: Starting from vertex 0
Step 2: Added vertex 0 to MST
Updated edge 0 - 1 (weight 10)
Updated edge 0 - 2 (weight 6)
Updated edge 0 - 3 (weight 5)
MST after step 2: 0-1 0-2 0-3
Step 3: Added vertex 3 to MST
Updated edge 3 - 2 (weight 4)
MST after step 3: 0-1 3-2 0-3
Step 4: Added vertex 2 to MST
MST after step 4: 0-1 3-2 0-3
Final MST: 0-1 3-2 0-3
Total weight: 19
```

Lab Report 17: Design and Analysis of Dijkstra's Algorithm

Date: May 08, 2025

Day: Thursday

1. Theory:

Dijkstra's Algorithm is a greedy algorithm used to find the shortest path from a single source vertex to all other vertices in a connected, undirected or directed graph with non-negative weighted edges. It starts from a chosen source vertex, explores adjacent vertices, and iteratively selects the unvisited vertex with the minimum tentative distance, updating distances as better paths are found. The result is a shortest-path tree from the source to all reachable vertices.

2. Key Concepts:

- It maintains a distance array to track the shortest known distance from the source to each vertex, initially set to infinity except for the source (set to 0).
- A priority queue or simple search is used to select the unvisited vertex with the smallest tentative distance.
- It relaxes edges by updating distances if a shorter path through the current vertex is found.
- Works only for graphs with non-negative edge weights to avoid incorrect results from negative cycles.

3. Properties of Dijkstra's Algorithm:

- Produces the shortest path from the source to all vertices if all edge weights are non-negative.
- Greedy in nature, always choosing the next vertex with the minimum current distance.
- Commonly used in routing (e.g., network packet routing, GPS navigation).
- Fails with negative weights; Bellman-Ford is an alternative for such cases.

4. Algorithm:

Algorithm: Dijkstra ShortestPath (graph, V, src)

Input: A graph with V vertices represented as an adjacency matrix, source vertex src

Output: Shortest distances from src to all vertices and the path tree

1. Initialize distance[V] with infinity (shortest distance to each vertex), parent[V] with -1 (to store the shortest path tree), and visited[V] with false.
 2. Set distance[src] = 0 (distance to source is 0).
 3. For count = 0 to V-1:
 - Find the unvisited vertex u with the minimum distance[u].
 - Mark visited[u] = true.
 - For each vertex v adjacent to u:
 - If v is unvisited and graph[u][v] is non-zero (edge exists) and distance[u] + graph[u][v] < distance[v]:
 - Update distance[v] = distance[u] + graph[u][v].
 - Set parent[v] = u.
 4. Return distance array (shortest distances) and parent array (for path reconstruction).
-

5. Time and Space Complexity Analysis:

Time Complexity:

- $O(V^2)$ – Using an adjacency matrix and linear search to find the minimum distance vertex (as implemented here for simplicity).
- Can be improved to $O((V + E) \log V)$ with a priority queue (e.g., min-heap) for selecting the minimum distance vertex.

Space Complexity:

- $O(V)$ – Space for distance, parent, and visited arrays.
- Additional $O(V)$ if storing paths explicitly.

6. Code Implementation:

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

// Function to find the unvisited vertex with minimum distance
int findMinDistance(int distance[], bool visited[], int V) {
    int minDist = 999999, minVertex = -1;
    for (int v = 0; v < V; v++) {
        if (!visited[v] && distance[v] < minDist) {
            minDist = distance[v];
            minVertex = v;
        }
    }
    return minVertex;
}

// Function to print the path from source to a vertex
void printPath(int parent[], int vertex, int src) {
    if (vertex == src) {
```

```

        cout << src;
        return;
    }
    if (parent[vertex] == -1) {
        cout << "No path exists";
        return;
    }
    printPath(parent, parent[vertex], src);
    cout << " <- " << vertex;
}

// Function to perform Dijkstra's Algorithm with step-by-step output
void dijkstra(int graph[][100], int V, int src) {
    int distance[V]; // Shortest distance from src to each vertex
    int parent[V];   // Parent array for shortest path tree
    bool visited[V]; // Track visited vertices
    int step = 1;

    // Step 1: Initialize distances and arrays
    cout << "Step " << step++ << ": Initializing distances" << endl;
    for (int i = 0; i < V; i++) {
        distance[i] = 999999; // Infinity
        visited[i] = false;
        parent[i] = -1;
    }
    distance[src] = 0;
    cout << "Distance from source " << src << ": ";
    for (int i = 0; i < V; i++)
        cout << "To " << i << " = " << (distance[i] == 999999 ? "INF" :
to_string(distance[i])) << " ";
    cout << endl;

    // Step 2: Process vertices
    for (int count = 0; count < V - 1; count++) {
        int u = findMinDistance(distance, visited, V);
        if (u == -1) break;

        // Mark vertex as visited
        visited[u] = true;
        cout << "Step " << step++ << ": Selected vertex " << u << " with distance " <<
distance[u] << endl;

        // Relax adjacent vertices
        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] && distance[u] != 999999 && distance[u] +
graph[u][v] < distance[v]) {
                cout << " Relaxing edge " << u << " - " << v << " (weight " << graph[u][v]
<< "): ";
                distance[v] = distance[u] + graph[u][v];
                parent[v] = u;
                cout << "Updated distance to " << v << " = " << distance[v] << endl;
            }
        }
    }

    // Print final shortest distances and paths
    cout << "Step " << step++ << ": Final shortest distances from source " << src << endl;
    for (int i = 0; i < V; i++) {
        cout << "Vertex " << i << ": Distance = " << (distance[i] == 999999 ? "INF" :
to_string(distance[i]));
        if (i != src) {
            cout << ", Path = ";
            printPath(parent, i, src);
        }
        cout << endl;
    }
}

```

```

int main() {
    int V, src;
    int graph[100][100]; // Adjacency matrix for graph

    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the adjacency matrix (enter 0 for no edge):" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << "Edge " << i << " to " << j << " weight: ";
            cin >> graph[i][j];
        }
    }
    cout << "Enter the source vertex: ";
    cin >> src;

    cout << "Graph edges:" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (graph[i][j] != 0)
                cout << i << " - " << j << ": " << graph[i][j] << endl;
        }
    }

    // Perform Dijkstra's Algorithm
    dijkstra(graph, V, src);

    return 0;
}

```

7. Output:

```

+ Dijkstra.cpp -o Dijkstra } ; if ($?) { .\Dijkstra }
Enter the number of vertices: 3
Enter the adjacency matrix (enter 0 for no edge):
Edge 0 to 0 weight: 0
Edge 0 to 1 weight: 5
Edge 0 to 2 weight: 6
Edge 1 to 0 weight: 8
Edge 1 to 1 weight: 9
Edge 1 to 2 weight: 4
Edge 2 to 0 weight: 5
Edge 2 to 1 weight: 7
Edge 2 to 2 weight: 6
Enter the source vertex: 3
Graph edges:
0 - 1: 5
0 - 2: 6
1 - 0: 8
1 - 1: 9
1 - 2: 4
2 - 0: 5
2 - 1: 7
2 - 2: 6
Step 1: Initializing distances
Distance from source 3: To 0 = INF To 1 = INF To 2 = INF
Step 2: Final shortest distances from source 3
Vertex 0: Distance = INF, Path = No path exists
Vertex 1: Distance = INF, Path = No path exists
Vertex 2: Distance = INF, Path = No path exists

```

Lab Report 18: Design and Analysis of Matrix Chain Multiplication

Date: May 08, 2025

Day: Thursday

Theory:

Matrix Chain Multiplication is a dynamic programming algorithm used to determine the most efficient way to multiply a sequence of matrices. Given a chain of matrices A_1, A_2, \dots, A_n , where each matrix A_i has dimensions $p_{i-1} \times p_i$, the goal is to parenthesize the sequence to minimize the total number of scalar multiplications. The order of multiplication affects the cost, as matrix multiplication is associative but not commutative.

Key Concepts:

- The problem uses a dynamic programming approach to break the chain into optimal subproblems.
- A table $m[i][j]$ stores the minimum number of scalar multiplications needed to compute the product of matrices from A_i to A_j .
- A table $s[i][j]$ tracks the optimal split point (k) to reconstruct the parenthesization.
- The cost of multiplying two matrices $A (p \times q)$ and $B (q \times r)$ is $p \times q \times r$ scalar multiplications.

Properties of Matrix Chain Multiplication:

- Finds the optimal parenthesization to minimize computation cost, not the actual product.
- Assumes matrices are compatible (i.e., the number of columns of one equals the number of rows of the next).
- Used in graphics, scientific computing, and optimization problems where matrix operations are frequent.
- The solution is not unique if multiple parenthesizations yield the same minimum cost.

Algorithm:

Algorithm: Matrix Chain Multiplication (p, n)

Input: Array $p[]$ of size $n+1$ containing dimensions (matrix A_i is $p_{i-1} \times p_i$), number of matrices n

Output: Minimum scalar multiplications and optimal parenthesization

1. Initialize $m[n][n]$ with 0 for diagonal (single matrix, no multiplication).

2. For length $l = 2$ to n :
o For $i = 1$ to $n - l + 1$:
▪ Set $j = i + l - 1$ (end of subchain).
▪ Set $m[i][j] = \text{infinity}$ (initial minimum cost).
▪ For $k = i$ to $j - 1$:
▪ Compute cost = $m[i][k] + m[k+1][j] + p[i-1] \times p[k] \times p[j]$.
▪ If cost < $m[i][j]$:
▪ Update $m[i][j] = \text{cost}$.
▪ Set $s[i][j] = k$ (optimal split point).
3. Return $m[1][n]$ (minimum cost for full chain) and s (for parenthesization).

Time and Space Complexity Analysis:

Time Complexity:

- $O(n^3)$ – Three nested loops: one for chain length (l), one for start index (i), and one for split point (k).
 - Computes all possible subproblems to find the optimal solution.
- Space Complexity:

- $O(n^2)$ – Space for the m and s tables, each of size $n \times n$, to store costs and split points.

Code Implementation:

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

// Function to print the optimal parenthesization
void printParenthesis(int s[][100], int i, int j, char& name) {
    if (i == j) {
        cout << name++;
        return;
    }
    cout << "(";
    printParenthesis(s, i, s[i][j], name);
    printParenthesis(s, s[i][j] + 1, j, name);
    cout << ")";
}

// Function to perform Matrix Chain Multiplication with step-by-step output
void matrixChainMultiplication(int p[], int n) {
    int m[100][100]; // Table for minimum scalar multiplications
    int s[100][100]; // Table for split points
    int step = 1;

    // Step 1: Initialize tables
    cout << "Step " << step++ << ": Initializing table for single matrices" << endl;
    for (int i = 1; i <= n; i++) {
        m[i][i] = 0; // No multiplications for single matrix
    }

    // Step 2: Fill tables for chains of increasing length
    for (int l = 2; l <= n; l++) {
        cout << "Step " << step++ << ": Computing chains of length " << l << endl;
```



```

        for (int i = 1; i <= n - 1 + 1; i++) {
            int j = i + 1 - 1;
            m[i][j] = 999999; // Infinity
            cout << "    Considering chain A" << i << " to A" << j << ": ";

            for (int k = i; k < j; k++) {
                int cost = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                cout << "Try split at k=" << k << " (cost = " << cost << "), ";
                if (cost < m[i][j]) {
                    m[i][j] = cost;
                    s[i][j] = k;
                    cout << "Updated min cost = " << m[i][j];
                }
                cout << endl;
            }
        }
    }

    // Step 3: Print the minimum cost and optimal parenthesization
    cout << "Step " << step++ << ": Final results" << endl;
    cout << "Minimum number of scalar multiplications: " << m[1][n] << endl;
    cout << "Optimal parenthesization: ";
    char name = 'A';
    printParenthesis(s, 1, n, name);
    cout << endl;

    // Print the m table for reference
    cout << "Cost table (m[i][j]):" << endl;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (m[i][j] == 999999)
                cout << "INF ";
            else
                cout << m[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int n;
    cout << "Enter the number of matrices: ";
    cin >> n;
    int p[100]; // Array to store dimensions
    cout << "Enter " << n + 1 << " dimensions (p0, p1, ..., pn):" << endl;
    for (int i = 0; i <= n; i++) {
        cout << "p" << i << ": ";
        cin >> p[i];
    }

    cout << "Matrix chain dimensions:" << endl;
    for (int i = 1; i <= n; i++) {
        cout << "Matrix A" << i << ": " << p[i - 1] << " x " << p[i] << endl;
    }

    // Perform Matrix Chain Multiplication
    matrixChainMultiplication(p, n);

    return 0;
}

```

Output:

```
trix_Chain_Multiplication.cpp -o Matrix_Chain_Multiplication } ; if ($?) { .\Matrix_Chain_Multiplication }
Enter the number of matrices: 4
Enter 5 dimensions (p0, p1, ..., pn):
p0: 40
p1: 20
p2: 30
p3: 40
p4: 50
Matrix chain dimensions:
Matrix A1: 40 x 20
Matrix A2: 20 x 30
Matrix A3: 30 x 40
Matrix A4: 40 x 50
Step 1: Initializing table for single matrices
Step 2: Computing chains of length 2
    Considering chain A1 to A2: Try split at k=1 (cost = 24000), Updated min cost = 24000
    Considering chain A2 to A3: Try split at k=2 (cost = 24000), Updated min cost = 24000
    Considering chain A3 to A4: Try split at k=3 (cost = 60000), Updated min cost = 60000
Step 3: Computing chains of length 3
    Considering chain A1 to A3: Try split at k=1 (cost = 56000), Updated min cost = 56000
    Try split at k=2 (cost = 72000),
    Considering chain A2 to A4: Try split at k=2 (cost = 90000), Updated min cost = 90000
    Try split at k=3 (cost = 64000), Updated min cost = 64000
Step 4: Computing chains of length 4
    Considering chain A1 to A4: Try split at k=1 (cost = 104000), Updated min cost = 104000
    Try split at k=2 (cost = 144000),
    Try split at k=3 (cost = 136000),
Step 5: Final results
Minimum number of scalar multiplications: 104000
Optimal parenthesization: (A((BC)D))
Cost table (m[i][j]):
0 24000 56000 104000
0 0 24000 64000
0 0 0 60000
0 0 0 0
```

Lab Report 19: Design and Analysis of 0/1 KnapSack

Date: May 08, 2025

Day: Thursday

Theory:

The 0-1 Knapsack Problem is a classic optimization problem solved using dynamic programming. Given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity, the goal is to determine which items to include (0 or 1, i.e., either take the item or not) to maximize the total value without exceeding the weight limit. Unlike the fractional knapsack, partial items are not allowed.

Key Concepts:

- Uses a 2D table $dp[i][w]$ to store the maximum value achievable with the first i items and a weight limit of w .
- For each item, decide to either include it (if weight allows) or exclude it, choosing the option that maximizes value.
- Builds solutions bottom-up, considering all items and weight capacities iteratively.
- Optimal substructure: the solution for a given i and w depends on solutions to smaller subproblems.

Properties of 0-1 Knapsack Problem:

- Guarantees the maximum value for the given capacity if all weights and values are non-negative.
- Does not allow fractional items, making it discrete and suited for dynamic programming.
- Used in resource allocation, cargo loading, and budgeting scenarios.
- NP-hard, but the dynamic programming approach provides a pseudo-polynomial solution.

Algorithm:

Algorithm: ZeroOne Knapsack (val , wt , W , n)

Input: Array $val[]$ of values, array $wt[]$ of weights, capacity W , number of items n

Output: Maximum achievable value and selected items

1. Initialize a 2D table $dp[n+1][W+1]$ with 0 (for base cases: no items or zero capacity).
-

2. For $i = 1$ to n :
o For $w = 0$ to W :
▪ If $wt[i-1] \leq w$ (item i can be included):
▪ $dp[i][w] = \max(dp[i-1][w], dp[i-1][w - wt[i-1]] + val[i-1])$.
▪ Else:
▪ $dp[i][w] = dp[i-1][w]$ (exclude item i).
3. Use $dp[n][W]$ for the maximum value.
4. Trace back through dp to find selected items: start at $dp[n][W]$, if $dp[i][w] \neq dp[i-1][w]$, include item i , reduce w by $wt[i-1]$, and repeat for $i-1$.

Time and Space Complexity Analysis:

Time Complexity:

- $O(n \times W)$ – Two nested loops: one for items (n) and one for weight capacity (0 to W).
- Pseudo-polynomial, as complexity depends on the numeric value of W .

Space Complexity:

- $O(n \times W)$ – Space for the 2D dp table to store maximum values for all subproblems.
- Can be optimized to $O(W)$ using a 1D array, but this version uses 2D for clarity.

Code Implementation:

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

// Function to solve 0-1 Knapsack with step-by-step output
void zeroOneKnapsack(int val[], int wt[], int W, int n) {
    int dp[100][100]; // 2D table for max value
    int step = 1;

    // Step 1: Initialize table
    cout << "Step " << step++ << ": Initializing DP table" << endl;
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
        }
    }

    // Step 2: Fill DP table
    for (int i = 1; i <= n; i++) {
        cout << "Step " << step++ << ": Considering item " << i << " (weight = " << wt[i-1]
        << ", value = " << val[i-1] << ")" << endl;
        for (int w = 0; w <= W; w++) {
            if (wt[i-1] <= w) {
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - wt[i-1]] + val[i-1]);
                cout << " Weight " << w << ": Include item " << i << "? Max(" << dp[i-1][w] << ", "
                << dp[i-1][w - wt[i-1]] + val[i-1] << ") = " << dp[i][w] << endl;
            } else {
                dp[i][w] = dp[i-1][w];
                cout << " Weight " << w << ": Cannot include item " << i << ", value = "
                << dp[i][w] << endl;
            }
        }
    }
}
```

```

    }
}

// Step 3: Print maximum value
cout << "Step " << step++ << ": Final maximum value" << endl;
cout << "Maximum value in knapsack: " << dp[n][W] << endl;

// Step 4: Trace back to find selected items
cout << "Step " << step++ << ": Tracing selected items" << endl;
cout << "Selected items: ";
int w = W, i = n;
while (i > 0 && w > 0) {
    if (dp[i][w] != dp[i-1][w]) {
        cout << "Item " << i << " (weight = " << wt[i-1] << ", value = " << val[i-1] <<
    " " << endl;
        w = w - wt[i-1];
        i--;
    } else {
        i--;
    }
}
cout << endl;

// Print DP table for reference
cout << "DP Table:" << endl;
for (int i = 0; i <= n; i++) {
    for (int w = 0; w <= W; w++) {
        cout << dp[i][w] << " ";
    }
    cout << endl;
}

int main() {
    int n, W;
    cout << "Enter the number of items: ";
    cin >> n;
    int val[100], wt[100]; // Arrays for values and weights
    cout << "Enter the values of " << n << " items:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Value of item " << i + 1 << ": ";
        cin >> val[i];
    }
    cout << "Enter the weights of " << n << " items:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Weight of item " << i + 1 << ": ";
        cin >> wt[i];
    }
    cout << "Enter the knapsack capacity: ";
    cin >> W;

    cout << "Items:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Item " << i + 1 << ": value = " << val[i] << ", weight = " << wt[i] <<
    endl;
    }

    // Perform 0-1 Knapsack
    zeroOneKnapsack(val, wt, W, n);

    return 0;
}

```

Output:

```
knapsack.cpp -o 01_knapsack } ; if ($?) { .\01_knapsack }
Enter the number of items: 3
Enter the values of 3 items:
Value of item 1: 5
Value of item 2: 9
Value of item 3: 6
Enter the weights of 3 items:
Weight of item 1: 8
Weight of item 2: 6
Weight of item 3: 5
Enter the knapsack capacity: 2
Items:
Item 1: value = 5, weight = 8
Item 2: value = 9, weight = 6
Item 3: value = 6, weight = 5
Step 1: Initializing DP table
Step 2: Considering item 1 (weight = 8, value = 5)
  Weight 0: Cannot include item 1, value = 0
  Weight 1: Cannot include item 1, value = 0
  Weight 2: Cannot include item 1, value = 0
Step 3: Considering item 2 (weight = 6, value = 9)
  Weight 0: Cannot include item 2, value = 0
  Weight 1: Cannot include item 2, value = 0
  Weight 2: Cannot include item 2, value = 0
Step 4: Considering item 3 (weight = 5, value = 6)
  Weight 0: Cannot include item 3, value = 0
  Weight 1: Cannot include item 3, value = 0
  Weight 2: Cannot include item 3, value = 0
Step 5: Final maximum value
Maximum value in knapsack: 0
Step 6: Tracing selected items
Selected items:
DP Table:
0 0 0
0 0 0
0 0 0
0 0 0
```

Lab Report 20: Design and Analysis of Floyd-Warshall Algorithm

Date: May 08, 2025

Day: Thursday

Theory:

The Floyd-Warshall Algorithm is a dynamic programming approach to find the shortest paths between all pairs of vertices in a weighted, directed graph, which may include negative edge weights but no negative cycles. It computes the shortest distances between every pair of vertices by considering each vertex as a potential intermediate point, iteratively improving the path lengths.

Key Concepts:

- Uses a 2D table $\text{dist}[i][j]$ to store the shortest distance from vertex i to vertex j .
- Initially, $\text{dist}[i][j]$ is set to the direct edge weight (or infinity if no edge exists), with $\text{dist}[i][i] = 0$.
- For each vertex k , it checks if a path through k reduces the distance between i and j .
- Handles both directed and undirected graphs (for undirected, edges are bidirectional).

Properties of Floyd-Warshall Algorithm:

- Finds all-pairs shortest paths in a graph, even with negative weights, as long as no negative cycles exist.
- Works for dense or sparse graphs, but is less efficient than Dijkstra's for sparse graphs with non-negative weights.
- Used in network routing, transit systems, and graph analysis (e.g., detecting transitive closures).
- Can reconstruct paths by tracking predecessors in a separate table.

Algorithm:

Algorithm: Floyd Warshall (graph, V)

Input: A graph with V vertices represented as an adjacency matrix, where $\text{graph}[i][j]$ is the weight of edge i to j (infinity if no edge)

Output: Shortest distances between all pairs and predecessor matrix for path reconstruction

1. Initialize $\text{dist}[V][V]$ with $\text{graph}[i][j]$ values; set $\text{dist}[i][i] = 0$.
 2. Initialize $\text{pred}[V][V]$ to track predecessors; set $\text{pred}[i][j] = i$ if edge exists, else -1.
 3. For $k = 0$ to $V-1$:
-

<ul style="list-style-type: none"> ○ For $i = 0$ to $V-1$: <ul style="list-style-type: none"> ▪ For $j = 0$ to $V-1$: <ul style="list-style-type: none"> ▪ If $\text{dist}[i][k] + \text{dist}[k][j] < \text{dist}[i][j]$: <ul style="list-style-type: none"> ▪ Update $\text{dist}[i][j] = \text{dist}[i][k] + \text{dist}[k][j]$. ▪ Set $\text{pred}[i][j] = \text{pred}[k][j]$.
4. Return dist (shortest distances) and pred (for path reconstruction).

Time and Space Complexity Analysis:

Time Complexity:

- $O(V^3)$ – Three nested loops iterate over all vertices (i, j, k) to compute shortest paths.
- Fixed complexity, regardless of graph density.

Space Complexity:

- $O(V^2)$ – Space for the dist and pred 2D matrices to store distances and predecessors.

Code Implementation:

```
#include<iostream>
using namespace std;
const int INF = 999999; // Infinity for no edge

// Function to print the path from i to j using predecessor matrix
void printPath(int pred[][100], int i, int j) {
    if (i == j) {
        cout << i;
        return;
    }
    if (pred[i][j] == -1) {
        cout << "No path";
        return;
    }
    printPath(pred, i, pred[i][j]);
    cout << " -> " << j;
}

// Function to perform Floyd-Warshall with step-by-step output
void floydWarshall(int graph[][100], int V) {
    int dist[100][100]; // Shortest distance matrix
    int pred[100][100]; // Predecessor matrix for path reconstruction
    int step = 1;

    // Step 1: Initialize distance and predecessor matrices
    cout << "Step " << step++ << ": Initializing distance and predecessor matrices" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
            if (i == j)
                pred[i][j] = -1;
            else if (graph[i][j] != INF && graph[i][j] != 0)
                pred[i][j] = i;
            else
                pred[i][j] = -1;
        }
    }
}
```



```

    }

    // Print initial distance matrix
    cout << "Initial distance matrix:" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                cout << "INF ";
            else
                cout << dist[i][j] << " ";
        }
        cout << endl;
    }

    // Step 2: Update distances using intermediate vertices
    for (int k = 0; k < V; k++) {
        cout << "Step " << step++ << ": Considering vertex " << k << " as intermediate" <<
endl;
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] <
dist[i][j]) {
                    cout << " Updating path " << i << " to " << j << " via " << k << ": ";
                    cout << dist[i][j] << " -> " << dist[i][k] + dist[k][j] << endl;
                    dist[i][j] = dist[i][k] + dist[k][j];
                    pred[i][j] = pred[k][j];
                }
            }
        }
    }

    // Step 3: Print final shortest distances and paths
    cout << "Step " << step++ << ": Final shortest distances" << endl;
    cout << "Shortest distances between all pairs:" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                cout << "INF ";
            else
                cout << dist[i][j] << " ";
        }
        cout << endl;
    }

    cout << "Paths between all pairs:" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << "Path from " << i << " to " << j << ": ";
            printPath(pred, i, j);
            cout << endl;
        }
    }
}

int main() {
    int V;
    int graph[100][100]; // Adjacency matrix for graph

    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the adjacency matrix (enter 999999 for no edge):" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << "Edge " << i << " to " << j << " weight: ";
            cin >> graph[i][j];
        }
    }
}

```

```

cout << "Graph edges:" << endl;
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
        if (graph[i][j] != INF && graph[i][j] != 0)
            cout << i << " -> " << j << ": " << graph[i][j] << endl;
    }
}

// Perform Floyd-Warshall Algorithm
floydWarshall(graph, V);

return 0;
}

```

Output:

```

C:\Users\firoj\... > Codes > main ?4 21:35 16.966s cd "c:\Users\firoj\
oyd_warshall.cpp -o Floyd_warshall } ; if ($?) { .\Floyd_warshall }
Enter the number of vertices: 2
Enter the adjacency matrix (enter 999999 for no edge):
Edge 0 to 0 weight: 5
Edge 0 to 1 weight: 9
Edge 1 to 0 weight: 4
Edge 1 to 1 weight: 6
Graph edges:
0 -> 0: 5
0 -> 1: 9
1 -> 0: 4
1 -> 1: 6
Step 1: Initializing distance and predecessor matrices
Initial distance matrix:
5 9
4 6
Step 2: Considering vertex 0 as intermediate
Step 3: Considering vertex 1 as intermediate
Step 4: Final shortest distances
Shortest distances between all pairs:
5 9
4 6
Paths between all pairs:
Path from 0 to 0: 0
Path from 0 to 1: 0 -> 1
Path from 1 to 0: 1 -> 0
Path from 1 to 1: 1

```

Lab Report 21: Design and Analysis of Subset Sum Problem

Date: May 08, 2025

Day: Thursday

Theory:

The Subset Sum Problem is a classic decision problem in computer science, often solved using dynamic programming. Given a set of positive integers and a target sum, the goal is to determine if there exists a subset of the given set whose elements sum exactly to the target. It is an NP-complete problem, but a pseudo-polynomial time solution exists via dynamic programming.

Key Concepts:

- Uses a 2D table $dp[i][s]$ to indicate whether a sum s is achievable using the first i elements.
- For each element, decide to include it or exclude it, checking if the target sum can be reached.
- Builds solutions bottom-up, considering all elements and possible sums up to the target.
- Optimal substructure: the solution for a given i and s depends on solutions to smaller subproblems.

Properties of Subset Sum Problem:

- Determines the existence of a subset summing to the target, not the subset itself (though traceable).
- Works with non-negative integers; negative values complicate the approach.
- Used in scheduling, cryptography, and resource allocation problems.
- NP-complete, but the dynamic programming solution is efficient for moderate sums.

Algorithm:

Algorithm: Subset Sum (arr, n, sum)

Input: Array arr[] of n positive integers, target sum

Output: Boolean indicating if a subset sums to the target, and the subset if it exists

1. Initialize a 2D table $dp[n+1][sum+1]$ with false; set $dp[i][0] = \text{true}$ (sum 0 is always achievable).
 2. For $i = 1$ to n :
 - For $s = 0$ to sum:
-

▪ If $\text{arr}[i-1] \leq s$:
▪ $\text{dp}[i][s] = \text{dp}[i-1][s] \text{ OR } \text{dp}[i-1][s - \text{arr}[i-1]]$ (include or exclude item i).
▪ Else:
▪ $\text{dp}[i][s] = \text{dp}[i-1][s]$ (exclude item i).
3. Check $\text{dp}[n][\text{sum}]$ for the result.
4. Trace back to find the subset: start at $\text{dp}[n][\text{sum}]$, if $\text{dp}[i][s] \neq \text{dp}[i-1][s]$, include $\text{arr}[i-1]$, reduce s by $\text{arr}[i-1]$, repeat for $i-1$.

Time and Space Complexity Analysis:

Time Complexity:

- $O(n \times \text{sum})$ – Two nested loops: one for items (n) and one for possible sums (0 to sum).
- Pseudo-polynomial, as complexity depends on the numeric value of sum .

Space Complexity:

- $O(n \times \text{sum})$ – Space for the 2D dp table to store subset sum possibilities.
- Can be optimized to $O(\text{sum})$ using a 1D array, but this version uses 2D for clarity.

Code Implementation:

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

// Function to solve Subset Sum with step-by-step output
void subsetSum(int arr[], int n, int sum) {
    bool dp[100][1000]; // 2D table for subset sum possibility
    int step = 1;

    // Step 1: Initialize table
    cout << "Step " << step++ << ": Initializing DP table" << endl;
    for (int i = 0; i <= n; i++) {
        for (int s = 0; s <= sum; s++) {
            if (s == 0)
                dp[i][s] = true; // Sum 0 is always achievable
            else if (i == 0)
                dp[i][s] = false; // No items, no sum
        }
    }
    cout << " Base case: Sum 0 is achievable for all i" << endl;

    // Step 2: Fill DP table
    for (int i = 1; i <= n; i++) {
        cout << "Step " << step++ << ": Considering item " << i << " (value = " << arr[i-1]
        << ")" << endl;
        for (int s = 1; s <= sum; s++) {
            if (arr[i-1] <= s) {
                dp[i][s] = dp[i-1][s] || dp[i-1][s - arr[i-1]];
                cout << " Sum " << s << ": Include item " << i << "? " << (dp[i-1][s] ?
                "true" : "false")
                << " OR " << (dp[i-1][s - arr[i-1]] ? "true" : "false") << " = " <<
                (dp[i][s] ? "true" : "false") << endl;
            } else {

```

```

        dp[i][s] = dp[i-1][s];
        cout << "    Sum " << s << ": Cannot include item " << i << ", result = " <<
(dp[i][s] ? "true" : "false") << endl;
    }
}

// Step 3: Print result
cout << "Step " << step++ << ": Final result" << endl;
if (dp[n][sum]) {
    cout << "Subset with sum " << sum << " exists!" << endl;
} else {
    cout << "No subset with sum " << sum << " exists!" << endl;
}

// Step 4: Trace back to find the subset
if (dp[n][sum]) {
    cout << "Step " << step++ << ": Tracing subset" << endl;
    cout << "Subset elements: ";
    int s = sum, i = n;
    while (i > 0 && s > 0) {
        if (dp[i][s] != dp[i-1][s]) {
            cout << arr[i-1] << " ";
            s = s - arr[i-1];
            i--;
        } else {
            i--;
        }
    }
    cout << endl;
}

// Print DP table for reference
cout << "DP Table:" << endl;
for (int i = 0; i <= n; i++) {
    for (int s = 0; s <= sum; s++) {
        cout << (dp[i][s] ? "T " : "F ");
    }
    cout << endl;
}

int main() {
    int n, sum;
    cout << "Enter the number of elements: ";
    cin >> n;
    int arr[100]; // Array for elements
    cout << "Enter the elements:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Element " << i + 1 << ": ";
        cin >> arr[i];
    }
    cout << "Enter the target sum: ";
    cin >> sum;

    cout << "Set elements:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Element " << i + 1 << ": " << arr[i] << endl;
    }

    // Perform Subset Sum
    subsetSum(arr, n, sum);

    return 0;
}

```

Output:

```
tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Enter the number of elements: 3
Enter the elements:
Element 1: 5
Element 2: 8
Element 3: 9
Enter the target sum: 2
Set elements:
Element 1: 5
Element 2: 8
Element 3: 9
Step 1: Initializing DP table
    Base case: Sum 0 is achievable for all i
Step 2: Considering item 1 (value = 5)
    Sum 1: Cannot include item 1, result = false
    Sum 2: Cannot include item 1, result = false
Step 3: Considering item 2 (value = 8)
    Sum 1: Cannot include item 2, result = false
    Sum 2: Cannot include item 2, result = false
Step 4: Considering item 3 (value = 9)
    Sum 1: Cannot include item 3, result = false
    Sum 2: Cannot include item 3, result = false
Step 5: Final result
No subset with sum 2 exists!
DP Table:
T F F
T F F
T F F
T F F
```

Lab Report 22: Design and Analysis of N-Queens Problem

Date: May 08, 2025

Day: Thursday

Theory:

The N-Queens Problem is a classic backtracking problem in computer science. The goal is to place N queens on an $N \times N$ chessboard such that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal. It's a constraint satisfaction problem, often solved using backtracking to explore all possible configurations.

Key Concepts:

- Each queen must be placed in a unique row and column to avoid horizontal and vertical attacks.
- Diagonal attacks are prevented by checking if the absolute difference of rows equals the absolute difference of columns for any two queens.
- Backtracking: Place a queen, check if the position is safe, proceed to the next row; if no solution is found, backtrack and try a different position.
- A solution exists for all $N \geq 4$; the problem tests algorithmic efficiency and constraint handling.

Properties of N-Queens Problem:

- Solutions are not unique; multiple valid placements exist for most N (e.g., 92 solutions for $N=8$).
- No solution exists for $N=2$ or $N=3$ due to the board size and attack constraints.
- Used in AI, constraint satisfaction, and scheduling problems requiring non-conflicting assignments.
- NP-complete in its generalized form, but backtracking provides a practical solution for small N.

Algorithm:

Algorithm: N_Queens (board, row, N)

Input: An $N \times N$ board, current row to place a queen, board size N

Output: True if N queens can be placed safely, with the board showing a solution

1. If row == N, all queens are placed successfully, return true.
 2. For col = 0 to N-1:
-

<ul style="list-style-type: none"> ○ Check if placing a queen at board[row][col] is safe: <ul style="list-style-type: none"> ▪ No other queen in the same column. ▪ No other queen in the left diagonal (check row-i, col-i for i > 0). ▪ No other queen in the right diagonal (check row-i, col+i for i > 0). ○ If safe, place queen (board[row][col] = 1). ○ Recursively try to place queens in the next row (N_Queens(board, row+1, N)). ○ If successful, return true. ○ If not, backtrack by removing the queen (board[row][col] = 0) and try the next column.
3. If no column works, return false.

Time and Space Complexity Analysis:

Time Complexity:

- $O(N!)$ – In the worst case, backtracking tries all possible placements, with N choices for the first row, N-1 for the second, and so on.
- Practical runtime is much faster due to early pruning of unsafe positions.

Space Complexity:

- $O(N^2)$ – Space for the $N \times N$ board to store queen positions.
- Additional $O(N)$ for recursion stack in the backtracking process.

Code Implementation:

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

// Function to check if a queen can be placed at board[row][col]
bool isSafe(int board[][100], int row, int col, int N) {
    // Check column above
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1)
            return false;
    }
    // Check upper-left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return false;
    }
    // Check upper-right diagonal
    for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
        if (board[i][j] == 1)
            return false;
    }
    return true;
}

// Function to solve N-Queens with step-by-step output
bool nQueens(int board[][100], int row, int N, int& step) {
    // Base case: all queens placed
```



```

    if (row == N) {
        cout << "Step " << step++ << ": Solution found!" << endl;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                cout << (board[i][j] == 1 ? "Q " : ". ");
            }
            cout << endl;
        }
        return true;
    }

    // Try placing queen in each column of current row
    cout << "Step " << step++ << ": Trying to place queen in row " << row << endl;
    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col, N)) {
            cout << "  Placing queen at (" << row << ", " << col << ")" << endl;
            board[row][col] = 1;

            // Recursively place queens in next rows
            if (nQueens(board, row + 1, N, step))
                return true;

            // Backtrack if placement fails
            cout << "Step " << step++ << ": Backtracking, removing queen from (" << row <<
            ", " << col << ")" << endl;
            board[row][col] = 0;
        }
    }
    cout << "Step " << step++ << ": No valid placement in row " << row << endl;
    return false;
}

int main() {
    int N;
    cout << "Enter the board size (N): ";
    cin >> N;
    int board[100][100] = {0}; // Initialize board with 0 (no queens)
    int step = 1;

    cout << "Initial board:" << endl;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << ". ";
        }
        cout << endl;
    }

    // Solve N-Queens
    if (!nQueens(board, 0, N, step)) {
        cout << "Step " << step++ << ": No solution exists for N = " << N << endl;
    }

    return 0;
}

```

Output:

```
main ?6 21:42 21.051s cd "c:\Users\firoj\OneDrive\
mpCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Enter the board size (N): 4
Initial board:
. . . .
. . . .
. . . .
. . . .

Step 1: Trying to place queen in row 0
  Placing queen at (0, 0)
Step 2: Trying to place queen in row 1
  Placing queen at (1, 2)
Step 3: Trying to place queen in row 2
Step 4: No valid placement in row 2
Step 5: Backtracking, removing queen from (1, 2)
  Placing queen at (1, 3)
Step 6: Trying to place queen in row 2
  Placing queen at (2, 1)
Step 7: Trying to place queen in row 3
Step 8: No valid placement in row 3
Step 9: Backtracking, removing queen from (2, 1)
Step 10: No valid placement in row 2
Step 11: Backtracking, removing queen from (1, 3)
Step 12: No valid placement in row 1
Step 13: Backtracking, removing queen from (0, 0)
  Placing queen at (0, 1)
Step 14: Trying to place queen in row 1
  Placing queen at (1, 3)
Step 15: Trying to place queen in row 2
  Placing queen at (2, 0)
Step 16: Trying to place queen in row 3
  Placing queen at (3, 2)
Step 17: Solution found!
. Q . .
. . . Q
Q . . .
. . Q .
```

Lab Report 23: Design and Analysis of Miller Rabin Primality Test

Date: May 08, 2025

Day: Thursday

Theory:

The Miller-Rabin Primality Test is a probabilistic algorithm used to determine whether a given number is prime. It leverages Fermat's Little Theorem and the properties of prime numbers to test if a number is composite, with a small probability of error. By repeatedly testing with random bases, it achieves high accuracy, making it efficient for large numbers where deterministic primality tests are too slow.

Key Concepts:

- Based on the fact that for a prime n , if a is a base ($1 < a < n$), then $a^{(n-1)} \equiv 1 \pmod{n}$ (Fermat's Little Theorem).
- Express $n-1$ as $d * 2^s$, where d is odd, and check if $a^d \equiv 1 \pmod{n}$ or $a^{(d * 2^r)} \equiv -1 \pmod{n}$ for some $r < s$.
- If neither condition holds, n is composite; otherwise, it's likely prime.
- Randomly select bases a to test; more tests reduce the error probability (e.g., $< 1/4^k$ for k tests).

Properties of Miller-Rabin Primality Test:

- Probabilistic: may falsely identify a composite as prime (a "strong liar"), but the error rate is low.
- Fast for large numbers, unlike trial division or deterministic methods like AKS.
- Commonly used in cryptography (e.g., RSA key generation) for testing large primes.
- For a composite n , at least $3/4$ of bases a reveal compositeness, making multiple iterations reliable.

Algorithm:

Algorithm: Miller Rabin Primality (n, k)

Input: Number n to test for primality, k iterations for accuracy

Output: "Probably prime" or "Composite"

1. If $n \leq 1$, return "Composite".
 2. If $n = 2$ or 3 , return "Probably prime".
 3. If n is even and > 2 , return "Composite".
 4. Write $n-1 = d * 2^s$, where d is odd and s is the power of 2.
-

5. For i = 1 to k:
○ Choose a random base a from 2 to n-2.
○ Compute $x = a^d \bmod n$.
○ If $x = 1$ or $x = n-1$, continue to next iteration.
○ For r = 0 to s-1:
▪ Compute $x = x^2 \bmod n$.
▪ If $x = n-1$, break (likely prime for this a).
▪ If $x = 1$, return “Composite”.
○ If $x \neq n-1$, return “Composite”.
6. Return “Probably prime”.

Time and Space Complexity Analysis:

Time Complexity:

- $O(k * \log^3 n)$ – Each iteration involves modular exponentiation ($O(\log n)$ for base, $O(\log n)$ for squaring), repeated k times; $\log n$ arises from bit operations.
- Fast in practice, especially for large n, with k controlling accuracy.

Space Complexity:

- $O(1)$ – Constant space for variables (a, d, s, x), excluding the input number.

Code Implementation:

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

// Function for modular exponentiation (base^exp mod n)
long long modPow(long long base, long long exp, long long n) {
    long long result = 1;
    base = base % n;
    cout << "    Computing " << base << "^" << exp << " mod " << n << ": ";
    while (exp > 0) {
        if (exp % 2 == 1)
            result = (result * base) % n;
        base = (base * base) % n;
        exp = exp / 2;
    }
    cout << result << endl;
    return result;
}

// Function for Miller-Rabin test for one base
bool millerTest(long long n, long long d, int& step) {
    // Pick a random base a from 2 to n-2
    long long a = 2 + rand() % (n - 3);
    cout << "    Testing base a = " << a << endl;

    // Compute a^d mod n
    long long x = modPow(a, d, n);
    cout << "    Step " << step++ << ": Initial x = a^d mod n = " << x << endl;
```

```

    if (x == 1 || x == n - 1)
        return true;

    // Repeatedly square x and check
    while (d != n - 1) {
        x = (x * x) % n;
        d *= 2;
        cout << "    Step " << step++ << ": Square x, now x = " << x << endl;
        if (x == 1)
            return false; // Composite
        if (x == n - 1)
            return true; // Likely prime
    }
    return false; // Composite
}

// Function to perform Miller-Rabin Primality Test
bool millerRabin(long long n, int k) {
    int step = 1;

    // Step 1: Handle base cases
    cout << "Step " << step++ << ": Checking base cases" << endl;
    if (n <= 1) {
        cout << "    n <= 1, composite" << endl;
        return false;
    }
    if (n == 2 || n == 3) {
        cout << "    n = " << n << ", probably prime" << endl;
        return true;
    }
    if (n % 2 == 0) {
        cout << "    n is even and > 2, composite" << endl;
        return false;
    }

    // Step 2: Decompose n-1 = d * 2^s
    cout << "Step " << step++ << ": Decomposing n-1 = d * 2^s" << endl;
    long long d = n - 1;
    int s = 0;
    while (d % 2 == 0) {
        d /= 2;
        s++;
    }
    cout << "    d = " << d << ", s = " << s << endl;

    // Step 3: Perform k iterations
    cout << "Step " << step++ << ": Performing " << k << " iterations" << endl;
    for (int i = 1; i <= k; i++) {
        cout << "Iteration " << i << ":" << endl;
        if (!millerTest(n, d, step)) {
            cout << "Step " << step++ << ": Number is composite" << endl;
            return false;
        }
    }

    // Step 4: Final result
    cout << "Step " << step++ << ": Final result" << endl;
    cout << "Number " << n << " is probably prime after " << k << " iterations" << endl;
    return true;
}

int main() {
    long long n;
    int k;
    cout << "Enter the number to test (n): ";
    cin >> n;
    cout << "Enter the number of iterations (k): ";

```

```

    cin >> k;

    // Seed random number generator
    srand(time(0));

    cout << "Testing primality of " << n << " with " << k << " iterations" << endl;

    // Perform Miller-Rabin Primality Test
    if (millerRabin(n, k)) {
        cout << "Result: " << n << " is probably prime" << endl;
    } else {
        cout << "Result: " << n << " is composite" << endl;
    }

    return 0;
}

```

Output:

```

... > Codes > main ?8 21:50 38.874s cd "c:\Users\firoj\OneDrive\Desktop\
miller_rabin_primality.cpp -o miller_rabin_primality } ; if ($?) { .\miller_rabin_primality }
Enter the number to test (n): 13
Enter the number of iterations (k): 4
Testing primality of 13 with 4 iterations
Step 1: Checking base cases
Step 2: Decomposing n-1 = d * 2^s
    d = 3, s = 2
Step 3: Performing 4 iterations
Iteration 1:
    Testing base a = 5
        Computing 5^3 mod 13: 8
        Step 4: Initial x = a^d mod n = 8
        Step 5: Square x, now x = 12
Iteration 2:
    Testing base a = 11
        Computing 11^3 mod 13: 5
        Step 6: Initial x = a^d mod n = 5
        Step 7: Square x, now x = 12
Iteration 3:
    Testing base a = 7
        Computing 7^3 mod 13: 5
        Step 8: Initial x = a^d mod n = 5
        Step 9: Square x, now x = 12
Iteration 4:
    Testing base a = 9
        Computing 9^3 mod 13: 1
        Step 10: Initial x = a^d mod n = 1
Step 11: Final result
Number 13 is probably prime after 4 iterations
Result: 13 is probably prime

```

Lab Report 24: Design and Analysis of Chinese Remainder Theorem

Date: May 08, 2025

Day: Thursday

Theory:

The Chinese Remainder Theorem (CRT) is a mathematical theorem used to solve a system of simultaneous linear congruences with pairwise coprime moduli. Given equations $x \equiv a_1 \pmod{m_1}$, $x \equiv a_2 \pmod{m_2}$, ..., $x \equiv a_n \pmod{m_n}$, where m_1, m_2, \dots, m_n are pairwise coprime, CRT guarantees a unique solution modulo $M = m_1 * m_2 * \dots * m_n$. It simplifies computations in modular arithmetic by breaking them into smaller, independent problems.

Key Concepts:

- Each congruence $x \equiv a_i \pmod{m_i}$ specifies a remainder a_i for x when divided by modulus m_i .
- Compute the product $M = m_1 * m_2 * \dots * m_n$, and for each i , calculate $M_i = M / m_i$ (partial product excluding m_i).
- Find the modular multiplicative inverse of M_i modulo m_i , i.e., y_i such that $M_i * y_i \equiv 1 \pmod{m_i}$.
- The solution is $x = a_1 * M_1 * y_1 + a_2 * M_2 * y_2 + \dots + a_n * M_n * y_n \pmod{M}$.

Properties of Chinese Remainder Theorem:

- Guarantees a unique solution modulo M if all moduli m_1, m_2, \dots, m_n are pairwise coprime ($\gcd(m_i, m_j) = 1$ for $i \neq j$).
- Widely used in cryptography, coding theory, and fast arithmetic in computer algebra.
- Reduces complex modular problems into simpler subproblems for each modulus.
- Solution is unique only modulo M ; adding multiples of M gives equivalent solutions.

Algorithm:

Algorithm: Chinese Remainder Theorem (a, m, n)

Input: Array $a[]$ of remainders, array $m[]$ of pairwise coprime moduli, number of equations n

Output: Solution x satisfying all congruences $x \equiv a_i \pmod{m_i}$

1. Compute $M = m[0] * m[1] * \dots * m[n-1]$ (product of all moduli).

2. For $i = 0$ to $n-1$:

 ○ Compute $M_i = M / m[i]$ (product of all moduli except $m[i]$).

<ul style="list-style-type: none"> Find y_i, the modular multiplicative inverse of M_i modulo $m[i]$, such that $M_i * y_i \equiv 1 \pmod{m[i]}$. <ul style="list-style-type: none"> Use Extended Euclidean Algorithm to solve for y_i.
3. Compute $x = a[0] * M_0 * y_0 + a[1] * M_1 * y_1 + \dots + a[n-1] * M_{n-1} * y_{n-1}$.
4. Return $x \bmod M$ (ensure the solution is within $[0, M)$).

Time and Space Complexity Analysis:

Time Complexity:

- $O(n * \log M)$ – Computing M takes $O(n)$; finding each modular inverse via Extended Euclidean Algorithm is $O(\log m_i)$, repeated n times; final computation involves multiplications and modulo.
- Depends on the size of moduli and efficiency of arithmetic operations.

Space Complexity:

- $O(n)$ – Space for arrays a , m , and intermediate values (M_i , y_i).
- Constant additional space for computations, excluding input.

Code Implementation:

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

// Function to find modular multiplicative inverse using Extended Euclidean Algorithm
long long modInverse(long long a, long long m, int& step) {
    long long m0 = m, t, q;
    long long x0 = 0, x1 = 1;
    cout << "    Step " << step++ << ": Finding inverse of " << a << " mod " << m << endl;

    if (m == 1) {
        cout << "    Step " << step++ << ": Modulus is 1, inverse = 0" << endl;
        return 0;
    }

    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m;
        a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
        cout << "        Quotient = " << q << ", a = " << a << ", m = " << m << ", x1 = " <<
x1 << endl;
    }

    if (x1 < 0)
        x1 += m0;
    cout << "    Step " << step++ << ": Inverse = " << x1 << endl;
    return x1;
}

// Function to solve system of congruences using Chinese Remainder Theorem
long long chineseRemainder(int a[], int m[], int n) {
```



```

int step = 1;

// Step 1: Compute product of all moduli
cout << "Step " << step++ << ": Computing product of moduli" << endl;
long long M = 1;
for (int i = 0; i < n; i++) {
    M *= m[i];
    cout << "    Multiply by m[" << i << "] = " << m[i] << ", M = " << M << endl;
}

// Step 2: Compute partial products and their inverses
cout << "Step " << step++ << ": Computing partial products and inverses" << endl;
long long x = 0;
for (int i = 0; i < n; i++) {
    long long Mi = M / m[i];
    cout << "    For i = " << i << ", M" << i << " = M / m[" << i << "] = " << Mi <<
endl;
    long long yi = modInverse(Mi, m[i], step);
    x += a[i] * Mi * yi;
    cout << "    Step " << step++ << ": Add a[" << i << "] * M" << i << " * y" << i
        << " = " << a[i] << " * " << Mi << " * " << yi << " = " << a[i] * Mi * yi <<
", x = " << x << endl;
}

// Step 3: Compute final result modulo M
cout << "Step " << step++ << ": Computing final result" << endl;
x = x % M;
if (x < 0)
    x += M;
cout << "    x mod M = " << x << " mod " << M << " = " << x << endl;

// Step 4: Return solution
cout << "Step " << step++ << ": Final solution" << endl;
cout << "Solution x = " << x << " (mod " << M << ")" << endl;
return x;
}

int main() {
    int n;
    cout << "Enter the number of congruences: ";
    cin >> n;
    int a[100], m[100]; // Arrays for remainders and moduli

    cout << "Enter the remainders (a[i]):" << endl;
    for (int i = 0; i < n; i++) {
        cout << "a[" << i << "]: ";
        cin >> a[i];
    }
    cout << "Enter the moduli (m[i], must be pairwise coprime):" << endl;
    for (int i = 0; i < n; i++) {
        cout << "m[" << i << "]: ";
        cin >> m[i];
    }

    cout << "System of congruences:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "x ≡ " << a[i] << " (mod " << m[i] << ")" << endl;
    }

    // Solve using Chinese Remainder Theorem
    long long result = chineseRemainder(a, m, n);
    cout << "Result: x = " << result << endl;

    return 0;
}

```

Output:

```
main 79 21:54 9.986s cd "c:\Users\firoj\OneDrive\De
nese_remainder_theorem.cpp -o chinese_remainder_theorem } ; if ($?) { .\chinese_remainder_
Enter the number of congruences: 3
Enter the remainders (a[i]):
a[0]: 2
a[1]: 3
a[2]: 2
Enter the moduli (m[i], must be pairwise coprime):
m[0]: 3
m[1]: 5
m[2]: 7
System of congruences:
x == 2 (mod 3)
x == 3 (mod 5)
x == 2 (mod 7)
Step 1: Computing product of moduli
Multiply by m[0] = 3, M = 3
Multiply by m[1] = 5, M = 15
Multiply by m[2] = 7, M = 105
Step 2: Computing partial products and inverses
For i = 0, M0 = M / m[0] = 35
Step 3: Finding inverse of 35 mod 3
Quotient = 11, a = 3, m = 2, x1 = 0
Quotient = 1, a = 2, m = 1, x1 = 1
Quotient = 2, a = 1, m = 0, x1 = -1
Step 4: Inverse = 2
Step 5: Add a[0] * M0 * y0 = 2 * 35 * 2 = 140, x = 140
For i = 1, M1 = M / m[1] = 21
Step 6: Finding inverse of 21 mod 5
Quotient = 4, a = 5, m = 1, x1 = 0
Quotient = 5, a = 1, m = 0, x1 = 1
Step 7: Inverse = 1
Step 8: Add a[1] * M1 * y1 = 3 * 21 * 1 = 63, x = 203
For i = 2, M2 = M / m[2] = 15
Step 9: Finding inverse of 15 mod 7
Quotient = 2, a = 7, m = 1, x1 = 0
Quotient = 7, a = 1, m = 0, x1 = 1
Step 10: Inverse = 1
Step 11: Add a[2] * M2 * y2 = 2 * 15 * 1 = 30, x = 233
Step 12: Computing final result
x mod M = 23 mod 105 = 23
Step 13: Final solution
Solution x = 23 (mod 105)
Result: x = 23
```