Ahsania Mission University of Science & Technology

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithm Sessional

Experiment No: 10

Experiment Date: 04.06.25

Submitted By:

Mehrin Nusrat Chowdhury

Roll: 1012320005101026

1st Batch, 2nd Year, 2nd Semester

Department of Computer Science and Engineering

Ahsania Mission University of Science & Technology

Submitted To:

Md. Fahim Faisal

Lecturer

Department of Computer Science and Engineering

Faculty of Engineering, Ahsania Mission University of Science & Technology

Submission Date: 27.06.25

Task 01: A C++ program that will implement Radix Sort Algorithm.

Theory:

Radix Sort is a non-comparative, integer-based sorting algorithm that sorts data with multiple digits by processing individual digits. It operates from the least significant digit (LSD) to the most significant digit (MSD), using a stable intermediate sorting algorithm such as Counting Sort to maintain the relative order of elements at each digit level.

Rather than comparing numbers directly, Radix Sort relies on digit-wise grouping, which allows it to achieve linear time complexity for a fixed number of digits. It's especially useful when sorting large sets of integers within a known digit range.

Algorithm Steps

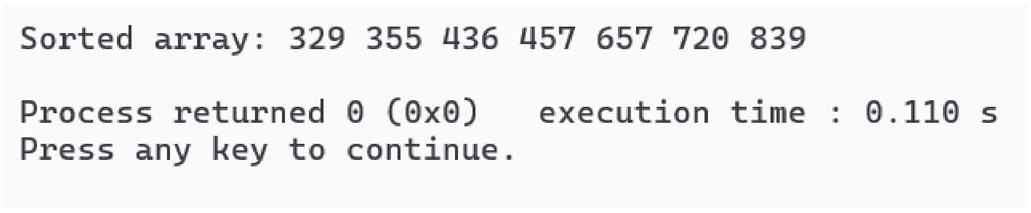
1. Identify the maximum value in the array to determine the number of digits.
2. Perform counting sort on the array for each digit, starting from the least significant digit.
3. Repeat the process for each subsequent digit by increasing the exponent by a power of 10 (i.e., 1s, 10s, 100s, etc.).
4. The array becomes sorted once all digits are processed.

Code:

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

int getMax(int arr[] , int n)
{
   int max = arr[0];
   for(int i = 1 ; i < n ; i++)
   {
      if(arr[i] > max)
      {
         max = arr[i];
      }
   }
   return max;
}

void countSort(int arr[] , int n , int exp)
{
   int output[n];
   int Count[10] = {0};
```

```
    for(int i = 0 ; i < n ; i++)
    {
        Count[(arr[i] / exp) % 10]++;
    }

    for(int i = 1 ; i < 10 ; i++)
    {
        Count[i] += Count[i - 1];
    }

    for(int i = n - 1 ; i >= 0 ; i--)
    {
        output[Count[(arr[i] / exp) % 10] -1] = arr[i];
        Count[(arr[i] / exp) % 10]--;
    }

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

void radixSort(int arr[] , int n)
{
    int max = getMax(arr, n);

    for(int exp = 1 ; max / exp > 0 ; exp *= 10)
    {
        countSort(arr , n , exp);
    }
}

void printArray(int arr[] , int n)
{
    for (int i = 0 ; i < n ; i++)
    {
        cout << arr[i] << " ";
    }
    cout << "\n";
}

int main()
{
    int arr[] = {329 , 457 , 657 , 839 , 436 , 720 , 355};
    int n = sizeof(arr) / sizeof(arr[0]);

    radixSort(arr , n);
    cout << "Sorted array: ";
    printArray(arr , n);

    return 0;
}
```

Output:



```
Sorted array: 329 355 436 457 657 720 839

Process returned 0 (0x0)   execution time : 0.110 s
Press any key to continue.
```

Conclusion:

Radix Sort efficiently organizes numeric data without direct comparisons, relying on digit-wise sorting and stability through Counting Sort. The algorithm excels in scenarios involving large volumes of numbers with consistent digit length, offering time-efficient performance. This lab exercise illustrates how algorithmic layering and stable sorting can lead to powerful and scalable solutions.

Task 02: A C++ program that will print Fibonacci Series using Memorization.

Theory:

The Fibonacci sequence is a classic mathematical series where each term is the sum of the two preceding ones, starting from 0 and 1. While the recursive method to compute Fibonacci numbers is conceptually simple, it suffers from excessive repeated calculations, especially for larger inputs, leading to exponential time complexity.

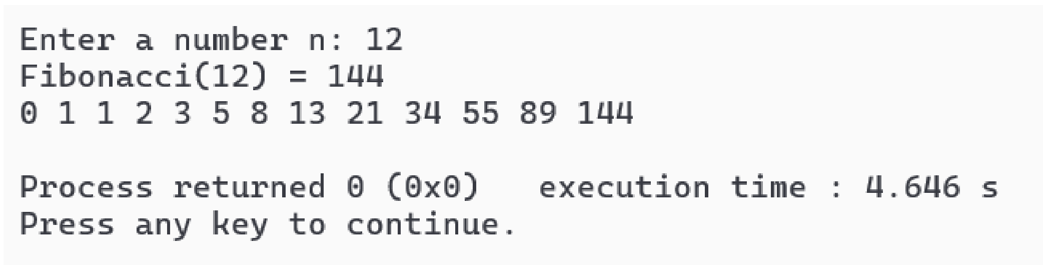
To address this inefficiency, memorization —a dynamic programming technique—is employed. In this approach, results of recursive subproblems are stored in an auxiliary array so that each value is computed only once. This drastically reduces the time complexity from exponential *$O(2^n)$* to linear *$O(n)$*, making the algorithm significantly faster and more memory-conscious for large values of *n*. The process demonstrates an efficient blend of recursion and optimization

Code:

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

int Fibonacci(int n , int memo[])
{
   if(n < 0)
   {
      cout << "Invalid input.";
      return -1;
   }

   if(n == 0)
   {
      return 0;
   }

   if(n == 1)
   {
      return 1;
   }

   if(memo[n] != -1)
   {
      return memo[n];
```

```
    }
    memo[n] = Fibonacci(n - 1 , memo) + Fibonacci(n - 2 , memo); // check for n-1 and n-2 as
well
    return memo[n];
}

void FibonacciSequence(int n , int memo[])
{
    if (n < 0)
    {
        cout << "Invalid input.";
        return;
    }

    for (int i = 0 ; i <= n ; i++)
    {
        cout << Fibonacci(i, memo) << " ";
    }
    cout << "\n";
}

int main()
{
    int n;
    cout << "Enter a number n: ";
    cin >> n;

    if (n < 0)
    {
        cout << "Invalid input.\n";
        return 0;
    }

    int memo[n + 1];
    for (int i = 0; i <= n; i++)
    {
        memo[i] = -1;
    }

    cout << "Fibonacci(" << n << ") = " << Fibonacci(n, memo) << "\n";

    FibonacciSequence(n , memo);

    return 0;

}
```

Output:



```
Enter a number n: 12
Fibonacci(12) = 144
0 1 1 2 3 5 8 13 21 34 55 89 144

Process returned 0 (0x0)   execution time : 4.646 s
Press any key to continue.
```

Conclusion:

The implementation of Fibonacci sequence using memorization validates the practical advantage of dynamic programming. By storing previously computed results, the algorithm eliminates redundant calculations and ensures a smooth and efficient computation of both individual Fibonacci values and the complete sequence up to *n*. This approach not only optimizes performance but also enhances code clarity and demonstrates the power of problem-solving through efficient algorithm design.