SCHOOL OF ENGINEERING & TECHNOLOGY

B.TECH. COMPUTER SCIENCE & ENGINEERING

DESIGN AND ANALYSIS OF ALGORITHMS

LAB FILE

AUG - DEC 2024

<u>SUBMITTED BY</u>

NAME OF STUDENT: TUSHAR GUPTA

ENROLLMENT NO.: 230542

SECTION: CSE-2

INDEX

11	**Fibonacci number & LCS using dynamic programming approach**		

Experiment-1

1. Aim

We have to wite a program to compare the time complexities of the following comparison- based sorting algorithms:
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort

We need to count the number of times the basic/primitive operations are being performed as a measure of the time complexity.
As, all the above algorithms are comparison-based sorting, the basic operation in this case is "comparison operation" of the elements.

2. Objective

The objective of this experiment is to empirically analyze and compare the time complexities of various comparison-based sorting algorithms.

3. Algorithm

1. **Initialize Random Seed**:
   - Set the random seed using srand(time(0)) to ensure different random numbers are generated in each run.
2. **Define Constants:** o Define SIZE as the size of each array (20 elements).
   - Define NUM_TRIALS as the number of trials (10 trials).
3. **Declare Arrays**: o Declare a 2D array arrays[NUM_TRIALS][SIZE] to store the random arrays for each trial.
   - Declare a 2D array results[6][NUM_TRIALS] to store the number of comparisons for each sorting algorithm across all trials.
4. **Fill Arrays with Random Numbers**: o For each trial, fill the corresponding array with random numbers between 0 and 99 using the fillArray function.
5. **Run Sorting Algorithms and Count Comparisons**:
   - For each trial, perform the following steps:
     - Copy the random array to a temporary array temp.
     - Sort temp using Bubble Sort and store the number of comparisons in results[0][trial].
     - Copy the random array to temp again.

- Sort temp using Insertion Sort and store the number of comparisons in results[1][trial].
- Repeat the above steps for Selection Sort, Merge Sort, Quick Sort, and Heap Sort, storing the number of comparisons in the respective rows of results.

6. **Display Results**: o Print the header row with the names of the sorting algorithms.
   - For each trial, print the trial number and the number of comparisons for each sorting algorithm.

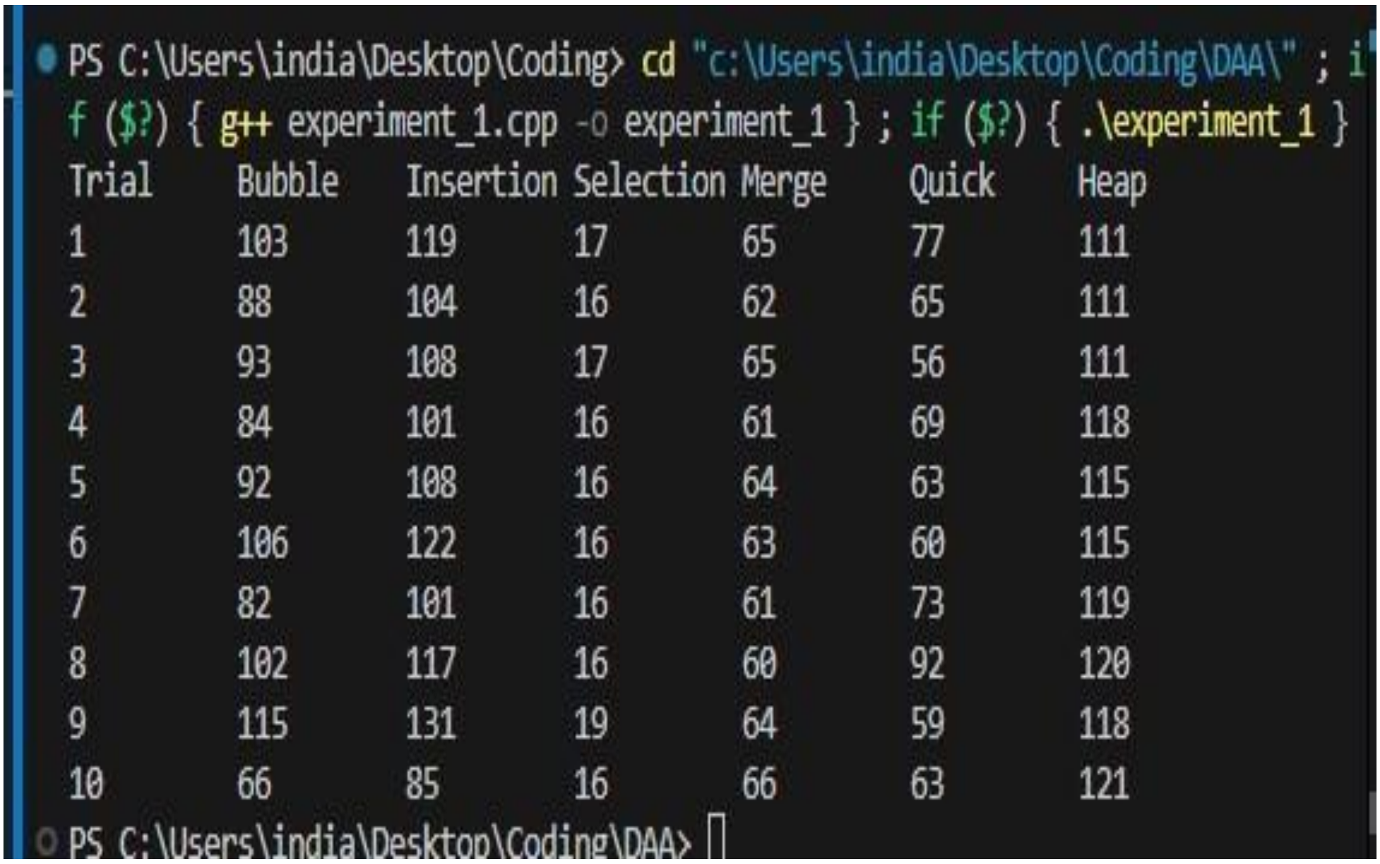
4.Implementation:

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

void fillArray(int arr[], int size)
  { for (int i = 0; i < size; i++) { arr[i]
     = rand() % 100;
  }
}

// 1. BUBBLE SORT
int BS(int arr[], int n) { int comp =
  0; for (int i = 0; i < n - 1; i++) {
  for (int j = 0; j < n - i - 1; j++)
      { if (arr[j] > arr[j + 1]) {
          int  temp  =  arr[j];
          arr[j]  =  arr[j  +  1];
          arr[j +
          1]     =     temp;
                comp++;
      }
```

```
        }
    }

    return comp;
}

// 2. INSERTION SORT
int IS(int arr[], int n) { int comp =
    0; for (int i = 1; i < n; i++) { int
    key = arr[i]; int j = i - 1; while (j
    >= 0 && arr[j] > key) { comp++;
    arr[j + 1] = arr[j]; j = j - 1;
        }
        if (j >= 0) comp++; // Count the last comparison
        arr[j + 1] = key;
    }

    return comp;
}


// 3. SELECTION SORT
int SS(int arr[], int n) { int comp
    = 0; for (int i = 0; i < n - 1; i++)
    { int minIndex = i; for (int j = i
    + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }

        if (minIndex != i) { int
              temp    = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex]
            =temp; comp++;
          }
}
        return comp;
    }


    // 4. MERGE SORT
        void merge(int arr[], int left, int mid, int right, int &comp)
        { int n1 = mid - left + 1;
        int n2 = right - mid;


        int   *L   =   new
        int[n1]; int *R =
        new int[n2];


        for (int i = 0; i < n1; i++) L[i] = arr[left + i];
        for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];


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


        while (i < n1) arr[k++] =
        L[i++]; while (j < n2) arr[k++]
        = R[j++];
        delete[] L; delete[]
        R;
    }

    void mergeSortHelper(int arr[], int left, int right, int
        &comp) { if (left < right) { int mid = left +
          (right - left) / 2; mergeSortHelper(arr, left,
          mid, comp); mergeSortHelper(arr, mid + 1,
          right, comp); merge(arr, left, mid, right,
          comp);
        }
    }


    int mergeSort(int arr[], int n) {
        int comp = 0;
        mergeSortHelper(arr, 0, n - 1,
        comp); return comp;
    }

    // 5. QUICK SORT
    int partition(int arr[], int low, int high, int &comp) {
        int pivot = arr[high]; int i =
        low - 1; for (int j = low; j <
        high; j++) { comp++;
          if (arr[j] < pivot)
            { i++;
            int temp =
            arr[i]; arr[i] = arr[j];
            arr[j] =
            temp;
          }
        }
        int temp = arr[i +
        1]; arr[i + 1] =
        arr[high]; arr[high]
        = temp; return i +
        1;

    }

    void quickSortHelper(int arr[], int low, int high, int
        &comp) { if (low < high) { int pi =
          partition(arr, low, high, comp);
          quickSortHelper(arr, low, pi - 1, comp);
          quickSortHelper(arr, pi + 1, high,
          comp);
        }
    }

    int quickSort(int arr[], int n) { int
                  comp = 0;
        quickSortHelper(arr, 0, n - 1,
        comp); return comp;
    }

    //6. HEAPIFY
    void heapify(int arr[], int n, int i, int &comp) {
        int largest = i; int
        left = 2 * i + 1; int
        right = 2 * i +
        2;

        if (left < n)
          {
          comp++;
          if (arr[left] > arr[largest])
           { largest = left; }
        }


        if  (right < n) { comp++; if
          (arr[right] > arr[largest])
                      { largest = right;}

        }


        if   (largest   !=   i)   {   swap(arr[i],
          arr[largest]);    heapify(arr,    n,
          largest, comp);
        }
    }

    int heapSort(int arr[], int n)
        { int comp = 0;


        for (int i = n / 2 - 1; i >= 0; i--)
                { heapify(arr, n, i, comp); }


        for (int i = n - 1; i >= 0; i--)
     {
          swap(arr[0],   arr[i]);
          heapify(arr,     i,     0,
          comp);
        }
    return comp;
    }

    int main() {
    srand(time(0));
```

```
// Size of each array
const int SIZE
= 20;

// Number of trials
const int NUM_TRIALS = 10;


int arrays[NUM_TRIALS][SIZE];

int results[6][NUM_TRIALS];


for (int i = 0; i < NUM_TRIALS; i++)
{ fillArray(arrays[i], SIZE) }


for (int trial = 0; trial < NUM_TRIALS; trial++) { int
   temp[SIZE];
            copy(begin(arrays[trial]), end(arrays[trial]),
  temp); results[0][trial] = BS(temp, SIZE);
  copy(begin(arrays[trial]), end(arrays[trial]),
  temp); results[1][trial] = IS(temp, SIZE);

   copy(begin(arrays[trial]), end(arrays[trial]), temp); results[2][trial]
   = SS(temp, SIZE);

   copy(begin(arrays[trial]), end(arrays[trial]), temp); results[3][trial]
   = mergeSort(temp, SIZE);

   copy(begin(arrays[trial]), end(arrays[trial]), temp); results[4][trial]
   = quickSort(temp, SIZE);


   copy(begin(arrays[trial]), end(arrays[trial]), temp); results[5][trial]
   = heapSort(temp, SIZE);
}
```

```
    cout << left << setw(10) <<
    "Trial"; cout << setw(10) <<
    "Bubble"; cout << setw(10) <<
    "Insertion"; cout << setw(10) <<
    "Selection"; cout << setw(10) <<
    "Merge"; cout << setw(10) <<
    "Quick";
    cout << setw(10) << "Heap" << endl;

    for (int i = 0; i <
        NUM_TRIALS; i++) {
        cout << setw(10) << (i +
        1); for (int j = 0; j < 6;
        j++)
        {     cout << setw(10) << results[j][i];
        } cout << endl;
    }
    return 0;
}
```



```
PS C:\Users\india\Desktop\Coding> cd "c:\Users\india\Desktop\Coding\DAA\" ; i
f ($?) { g++ experiment_1.cpp -o experiment_1 } ; if ($?) { .\experiment_1 }
Trial     Bubble    Insertion Selection Merge     Quick     Heap
1         103       119       17        65        77        111
2         88        104       16        62        65        111
3         93        108       17        65        56        111
4         84        101       16        61        69        118
5         92        108       16        64        63        115
6         106       122       16        63        60        115
7         82        101       16        61        73        119
8         102       117       16        60        92        120
9         115       131       19        64        59        118
10        66        85        16        66        63        121
PS C:\Users\india\Desktop\Coding\DAA> 
```

Experiment-2

1. Aim

To learn the use of Stack in the Recursion Algorithms.

2. Objective

We need to count the number of times the function is called in order to find the size of the stack.

3. Algorithm

We will write the normal code of the recursion and in that we will introduce the count variable and add it before the function is about to be called again in order to get the exact number of times the function was called.

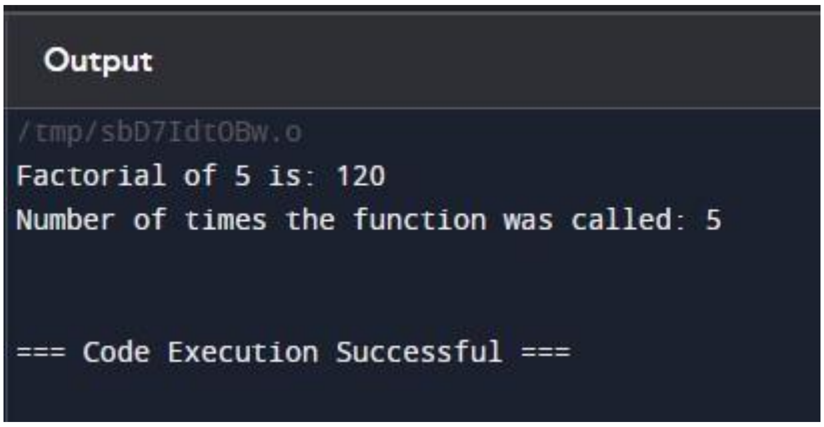
4. Implementation

1. FACTORIAL OF

NUMBER-

```
#include
```

```
<iostream>    using
namespace std;

int factorial(int n, int& count) {
    count++;

    if (n <=
        1)
        retur
        n 1;
    else return n * factorial(n
        - 1, count);
}

int
    main(
    ) { int
    numb
    er = 5;
    int
    count
    = 0;


    int result = factorial(number, count);

    cout << "Factorial of " << number << " is: " << result << endl;
    cout << "Number of times the function was called: " << count
    << endl;


    return 0;
}
```



```
/tmp/sbD7IdtOBw.o
Factorial of 5 is: 120
Number of times the function was called: 5


=== Code Execution Successful ===
```

2. Fibonacci

– #include

<iostream>

using

namespace

std;

int fibonacci(int n, int& count) {

count++;

if

(n

<

=

1)

re

t

u

r

n

n;

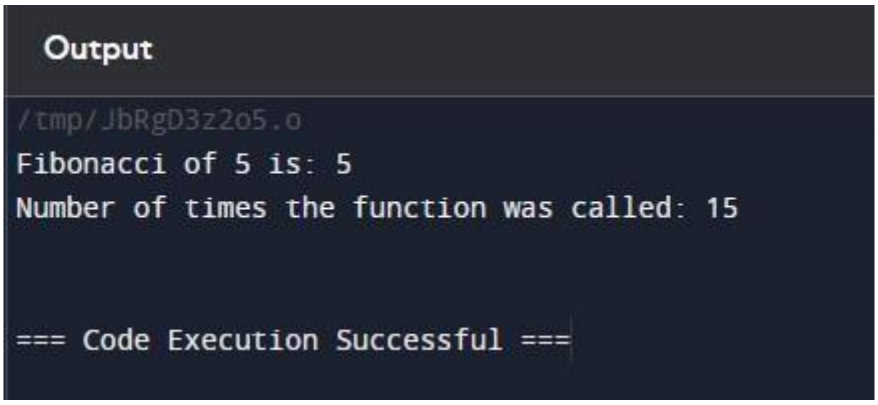
el

s

e

return fibonacci(n - 1, count) + fibonacci(n - 2, count);

```
}
int

   main(

   ) {  int

   numb

   er    =

   5;   int

   count

   = 0;


   int result = fibonacci(number, count);


   cout << "Fibonacci of " << number << " is: " << result <<
   endl;

   cout << "Number of times the function was called: " << count << endl;


   return 0;

}
```



Output

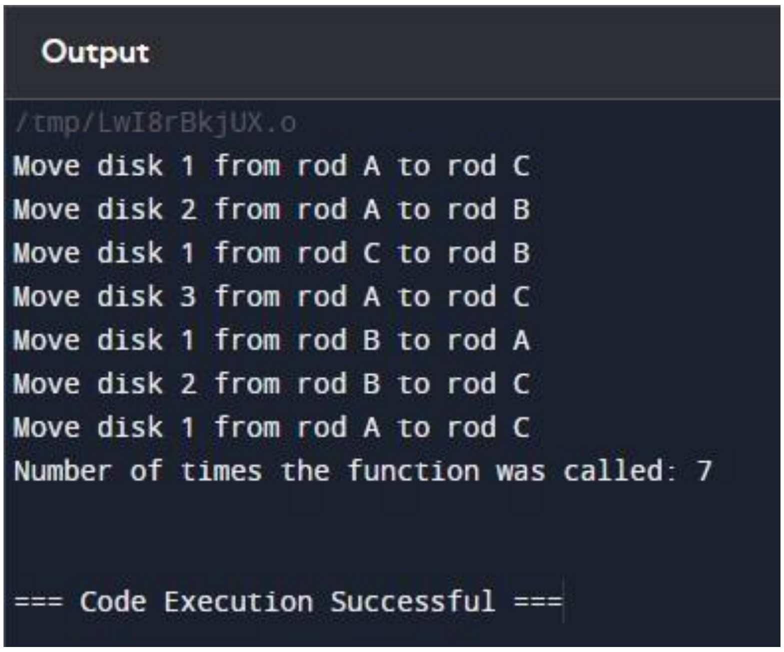
```
/tmp/JbRgD3z2o5.o
Fibonacci of 5 is: 5
Number of times the function was called: 15


=== Code Execution Successful ===
```

3. Tower Of

Hanoi

```
#include

<iostream>

using

namespace

std;
```

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod, int& count) {
   count++;


   if (n == 1) { cout << "Move disk 1 from rod " << from_rod << " to rod " <<
      to_rod << endl; return;
   }

   towerOfHanoi(n - 1, from_rod, aux_rod, to_rod, count);
   cout << "Move disk " << n << " from rod " << from_rod <<
   " to rod " << to_rod << endl; towerOfHanoi(n - 1, aux_rod,
   to_rod, from_rod, count);
}


int main() {
   int n = 3;
   int count = 0;


   towerOfHanoi(n, 'A', 'C', 'B', count);


   cout << "Number of times the function was called: " << count << endl;


   return 0;
}
```


Output

```
/tmp/LwI8rBkjUX.o
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
Number of times the function was called: 7


=== Code Execution Successful ===
```

Experiment-3

1. Aim:

To implement and understand the working of the Merge Sort and Binary Search algorithms
using the divide-and-conquer and decrease-and-conquer approaches, respectively.

2.Objective:

1. To gain hands-on experience in implementing recursive algorithms.
2. To understand the divide-and-conquer strategy through the Merge Sort algorithm.
3. To understand the decrease-and-conquer strategy through the Binary Search algorithm.
4. To analyze the time complexity and efficiency of these algorithms.
5. To develop problem-solving skills by applying these algorithms to practical problems

3.Algorithm:

Merge Sort Algorithm:

1. 2. 3. Divide: Split the array into two halves. Conquer: Recursively sort each half.

Combine: Merge the two sorted halves to produce the sorted array.

Binary Search Algorithm:

1. Input: Sorted array arr, target value target, low index low, high index high.
2. Base Case: If low > high, return -1 (target not found).
3. Calculate Mid: mid = (low + high) // 2.
4. Check Mid:
   - If arr[mid] == target, return mid.
   - If arr[mid] > target, search in the left half (low to mid-1).
   - If arr[mid] < target, search in the right half (mid+1 to high).

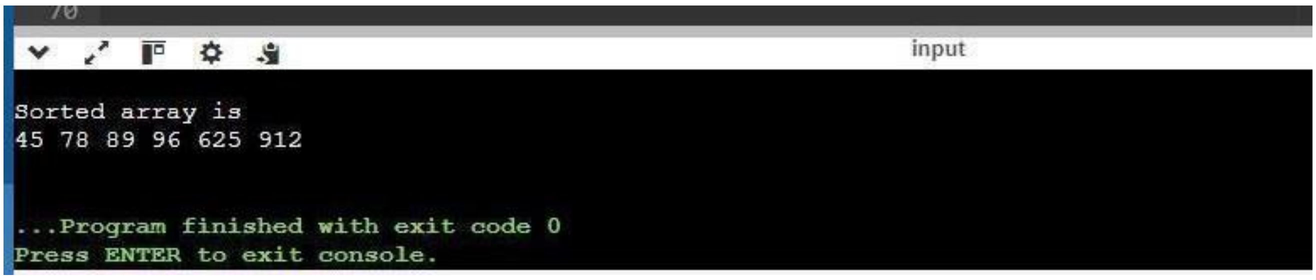
4.Implementation:

1.merge sort

```
#include <iostream> using

namespace std;

void merge(int arr[], int left, int mid, int right) { int

    n1 = mid - left + 1;
    int n2 = right - mid;




    int L[n1], R[n2];


    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];


    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {

    if (L[i] <= R[j]) { arr[k] =

    L[i]; i++;

        } else { arr[k]

            = R[j]; j++;

        }

        k++;

    }
}

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

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

void mergeSort(int arr[], int left, int right) { if
    (left < right) { int mid = left + (right
        - left) / 2;


        mergeSort(arr,  left, mid); mergeSort(arr,
        mid + 1, right);


         merge(arr, left, mid, right);
    }
}

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

int        main()        {        int        arr[]        =
    {625,78,45,89,96,912};     int     arr_size     =
    sizeof(arr) / sizeof(arr[0]); cout << "Given
    array is \n"; printArray(arr, arr_size);
```

```
mergeSort(arr, 0, arr_size - 1);

        cout << "\nSorted array is \n";
        printArray(arr, arr_size); return
        0;
    }
```

For Merge Sort

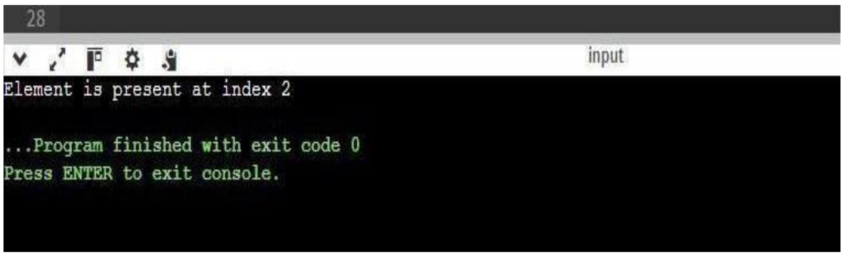


```
Sorted array is
45 78 89 96 625 912


...Program finished with exit code 0
Press ENTER to exit console.
```

2.binary search

```
  #include <iostream>


int binarySearch(int arr[], int left, int right, int x) { if
  (right >= left) { int mid =  left  +
     (right - left) / 2;


     if (arr[mid] == x)
        return mid;


     if (arr[mid] > x)
       return binarySearch(arr, left, mid - 1, x);


     return binarySearch(arr, mid + 1, right, x);
  }


  return -1;
}


int main() {
  int arr[] = {2, 3, 4, 10, 40}; int n =
  sizeof(arr) / sizeof(arr[0]); int x =
  10;
  int result = binarySearch(arr, 0, n - 1, x);
  (result == -1) ? std::cout << "Element is not present in array"
                 : std::cout << "Element is present at index " << result; return 0;
}
```

For Binary Search



```
Element is present at index 2

...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment-4

1. Aim

To implement and compare two algorithms for finding the minimum and maximum
elements in an unsorted array: an iterative brute-force approach and a recursive divide-and-conquer
approach

2. Objective

1. Implement an iterative brute-force function to find the minimum and maximum elements in an array.
2.Implement a recursive divide-and-conquer function to find the minimum and maximum elements in an array.
3.Compare the number of comparisons made by each function over 10 test runs with varying array sizes.
4.Tabulate the results to analyze the efficiency of both approaches.

3. Algorithm

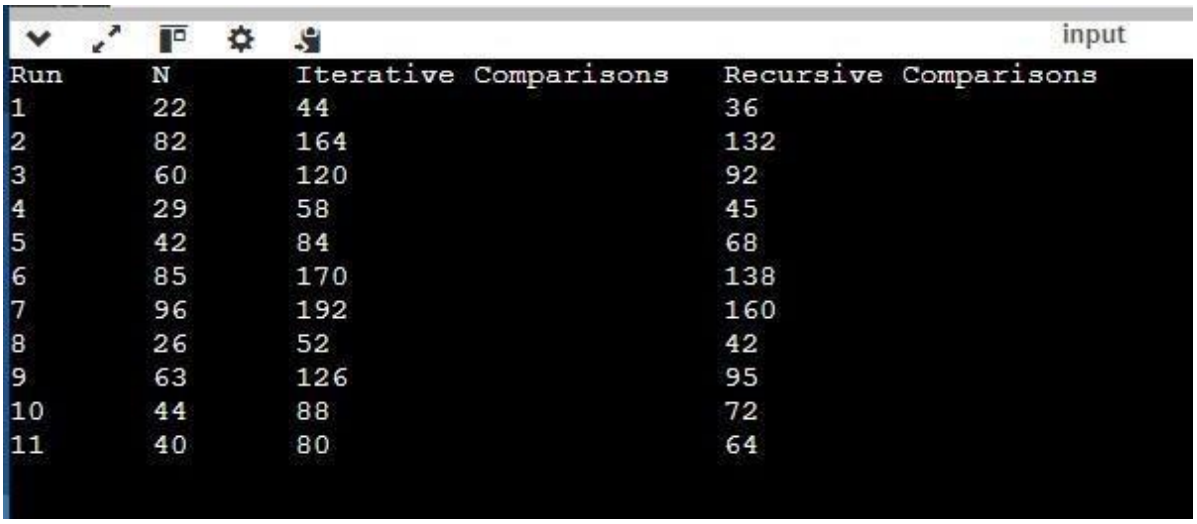
1. Initialize min to a very large value and max to a very small value.
2.Iterate through each element of the array.

3.Update min if the current element is smaller than min.
4.Update max if the current element is larger than max.
5. Return min and max.

4.Implementation

```
#include <iostream>

#include  <cstdlib>   #include

<ctime>

#include        <climits>

#include   <algorithm>

using namespace

std;

void findMinMaxIterative(int arr[], int n, int &min, int &max, int &comparisons) { min =

    INT_MAX; max = INT_MIN;

    comparisons = 0;


    for (int i = 0; i < n; ++i) { if

        (arr[i] < min) {

            min = arr[i];

        }

        if (arr[i] > max) {

            max = arr[i];

        }

        comparisons += 2;

    }

}

        void findMinMaxRecursive(int arr[], int low, int high, int &min, int &max, int &comparisons) { if

    (low == high) { if (arr[low] < min) min =

        arr[low];  if  (arr[low]  >  max)  max  =

        arr[low]; comparisons += 2;

        return;

    }
```

```
if (high == low + 1) { if (arr[low] < arr[high]) { if
        (arr[low] < min) min = arr[low]; if
        (arr[high] > max) max = arr[high];
            } else { if (arr[high] < min) min =
                arr[high];
                if (arr[low] > max) max = arr[low];
            }
            comparisons +=
            3; return;
        }

int mid = (low + high) / 2; findMinMaxRecursive(arr, low, mid, min, max,
        comparisons); findMinMaxRecursive(arr, mid + 1, high, min, max,
        comparisons);
    }


    void findMinMaxIterative(int arr[], int n, int &min, int &max, int &comparisons);
        void findMinMaxRecursive(int arr[], int low, int high, int &min, int &max, int &comparisons);

int main() {
    srand(time(0));
        const int testRuns = 11;


        cout << "Run\tN\tIterative Comparisons\tRecursive Comparisons\n";
        for (int run = 1; run <= testRuns; ++run) { int N
            = rand() % 100 + 1;
            int arr[N];
            for (int i = 0; i < N; ++i) {
                arr[i] = rand() % 1000;
            }


            int minIter, maxIter, comparisonsIter;
            findMinMaxIterative(arr, N, minIter, maxIter, comparisonsIter);
            int minRec = INT_MAX, maxRec = INT_MIN, comparisonsRec = 0;
            findMinMaxRecursive(arr, 0, N - 1, minRec, maxRec, comparisonsRec);


                cout << run << "\t" << N << "\t" << comparisonsIter << "\t\t\t" << comparisonsRec << "\n";
        }

        return 0;
    }
```



```
Run     N       Iterative Comparisons    Recursive Comparisons
1       22      44                       36
2       82      164                      132
3       60      120                      92
4       29      58                       45
5       42      84                       68
6       85      170                      138
7       96      192                      160
8       26      52                       42
9       63      126                      95
10      44      88                       72
11      40      80                       64
```

Experiment-5

1. Aim

1. To sort an unsorted array using the Quick Sort algorithm.
2.To compute (X^n) for (n > 0) using the Divide-and-Conquer approach

2. Objective

1.Understand the Quick Sort algorithm and its implementation.
2.Learn the Divide-and-Conquer approach for solving problems.
3.Implement the Quick Sort algorithm to sort an array in ascending order.
4.Implement the Divide-and-Conquer approach to compute (X^n).

3. Algorithm

1. Quick Sort Algorithm

Algorithm:

1.**Choose a Pivot**: Select a pivot element from the array.
2.**Partition**: Rearrange the array such that elements less than the pivot are on the left, elements greater than the pivot are on the right.
3.**Recursively Apply**: Apply the same process to the left and right subarrays.

2. Compute (X^n) using Divide-and-Conquer

Algorithm:

1. **Base Case**: If (n = 0), return 1.
2. **Recursive Case**:

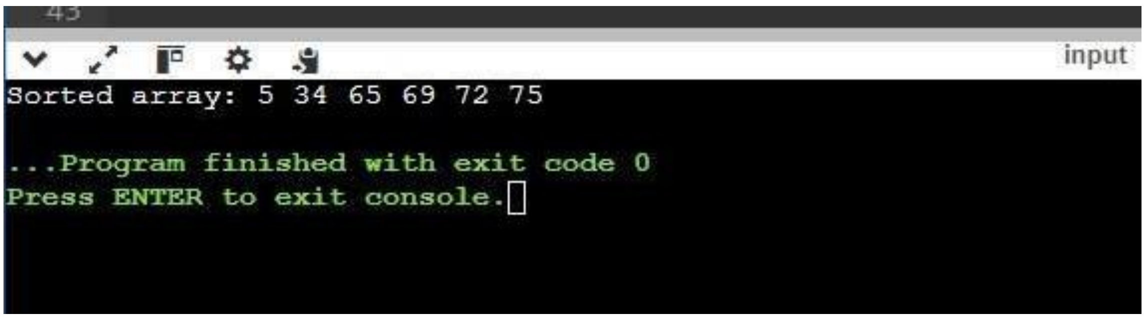
o If (n) is even, compute (X^{n/2}) and return ((X^{n/2}).2).
o If (n) is odd, compute (X^{(n-1)/2}) and return
(X \times(X^{(n-1)/2}).2)

4.Implementation:

1. quick sort

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

void swap(int* a, int* b) {
    int t = *a; *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high) { int pivot
    = arr[high];
    int i = (low - 1);

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

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

int main() { int arr[] = {65, 69, 75, 72,
  34,   5};   int   n   =   sizeof(arr)   /
  sizeof(arr[0]);  quickSort(arr,  0,  n  -
  1); cout << "Sorted array: "; for
   (int i = 0; i < n; i++) { cout
             << arr[i] << " ";
   }
   return 0;
}
```

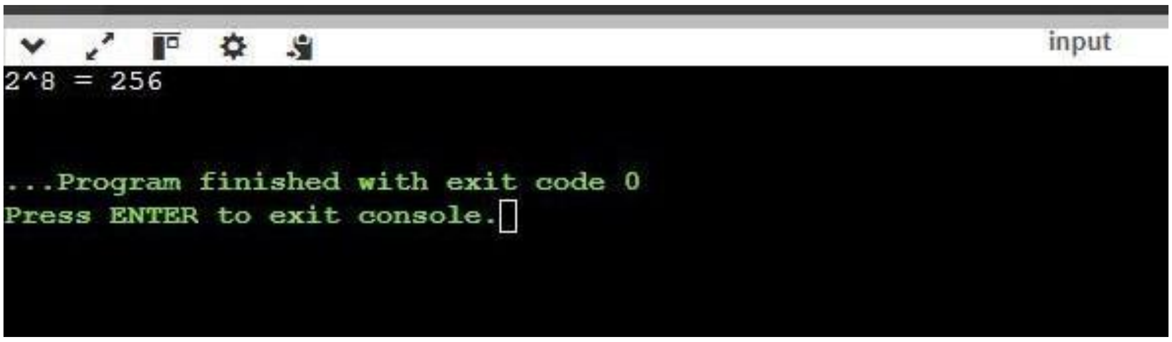


```
Sorted array: 5 34 65 69 72 75

...Program finished with exit code 0
Press ENTER to exit console.
```

2. power calculation :

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

double power(double x, int n) {
   if (n == 0) return 1;  double
   half = power(x, n / 2); if (n % 2
   == 0) return half * half; else
       return half * half * x;
}
```

```
int  main() { double
    x = 2.0;
    int n = 8; cout << x << "^" << n << " = " << power(x, n) << endl;
    return 0;
}
```



```
2^8 = 256

...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment-6

1. Aim

To efficiently multiply two large integers using the Karatsuba algorithm, which reduces the multiplication complexity compared to the traditional method and efficiently multiply two large square matrices using Strassen's algorithm, which reduces the number of multiplications compared to the traditional method**.**

2.Objective

Implement in C++.
Achieve a time complexity of (O(n^{\log_2 3}) \approx O(n^{1.585})) instead of the traditional (O(n^2)).

Achieve a time complexity of (O(n^{\log_2 7}) \approx O(n^{2.81})) instead of the traditional (O(n^3))Algorithm

3.Algorithm

1.Karatsuba's Algorithm

1. **Divide**: Split each number into two halves.
2. **Conquer**: Recursively compute three products of the smaller numbers.
3. **Combine**: Use the three products to construct the final result.

2.Strassen's Algorithm

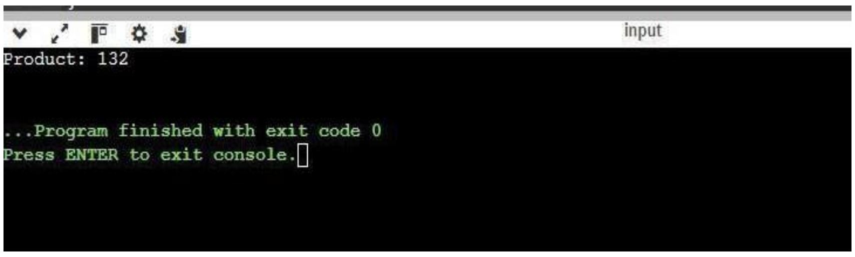
1. **Divide**: Split each matrix into four submatrices.
2. **Conquer**: Recursively compute seven products of the submatrices.
3. **Combine**: Use the seven products to construct the final result.

4. Implementation

1. Karatsuba's Algorithm

```
#include <iostream>

#include <string>

#include <algorithm>


using namespace std;

int makeEqualLength(string &str1, string &str2) { int

len1 = str1.size(); int len2 = str2.size();

    if (len1 < len2) { str1.insert(0,

        len2 - len1, '0'); return len2;

    } else if (len1 > len2) {

        str2.insert(0, len1 - len2, '0');

    }

    return len1;

}


string addBitStrings(const string &first, const string &second) { string

    result;

    int length = makeEqualLength(const_cast<string&>(first), const_cast<string&>(second));

    int carry = 0;


    for (int i = length - 1; i >= 0; i--) { int firstBit

        = first[i] - '0';

        int secondBit = second[i] - '0'; int sum = (firstBit ^ secondBit ^ carry) + '0';

        result.insert(0, 1, sum); carry = (firstBit & secondBit) | (secondBit &

        carry) | (firstBit & carry);

    }
```

```
        if (carry)  {  result.insert(0,
            1, '1');
        }

return result;
    }


    int multiplySingleBit(const string &a, const string &b) { return
        (a[0] - '0') * (b[0] - '0');
    }


    long long multiply(const string &X, const string &Y) {
        int n = makeEqualLength(const_cast<string&>(X), const_cast<string&>(Y));

if (n == 0) return 0;
        if (n == 1) return multiplySingleBit(X, Y);

int fh = n / 2;
        int sh = n - fh;

        string Xl = X.substr(0, fh);
        string  Xr  =  X.substr(fh,  sh);
        string Yl = Y.substr(0, fh);
        string Yr = Y.substr(fh, sh);

        long long P1 = multiply(Xl, Yl); long long P2 = multiply(Xr, Yr); long long
        P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr)); return P1 * (1LL
        << (2 * sh)) + (P3 - P1 - P2) * (1LL << sh) + P2;

    }

int main() { string num1 =

    "1100";
        string num2 = "1011";
    cout << "Product: " << multiply(num1, num2) << endl; return
    0;
}
```



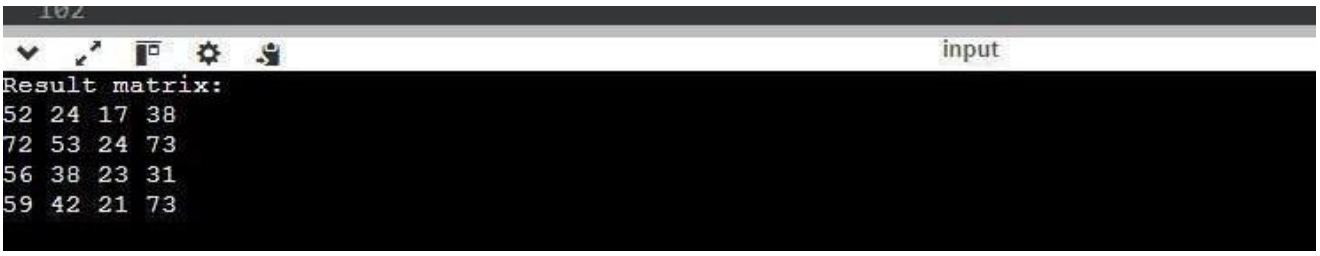
```
Product: 132

...Program finished with exit code 0
Press ENTER to exit console.
```

2. Strassen's Algorithm

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


typedef vector<vector<int>> matrix;


matrix add(const matrix &A, const matrix &B) { int n
    = A.size(); matrix C(n, vector<int>(n));
    for (int i = 0; i < n; ++i) for
        (int j = 0; j < n; ++j)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}

matrix subtract(const matrix &A, const matrix &B) { int n =
A.size(); matrix C(n, vector<int>(n));
    for (int i = 0; i < n; ++i) for
        (int j = 0; j < n; ++j)
            C[i][j] = A[i][j] - B[i][j];
    return C;
}


matrix strassen(const matrix &A, const matrix &B) {
    int n = A.size(); if
    (n == 1) {
        return {{A[0][0] * B[0][0]}};
    }

int  newSize   =   n   /   2;   matrix   A11(newSize,
    vector<int>(newSize));   matrix   A12(newSize,
    vector<int>(newSize));   matrix   A21(newSize,
    vector<int>(newSize));   matrix   A22(newSize,
    vector<int>(newSize));   matrix   B11(newSize,
    vector<int>(newSize));   matrix   B12(newSize,
    vector<int>(newSize));   matrix   B21(newSize,
    vector<int>(newSize));
    matrix B22(newSize, vector<int>(newSize));

    for (int i = 0; i < newSize; ++i) { for
        (int j = 0; j < newSize; ++j) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + newSize];
            A21[i][j] = A[i + newSize][j];
            A22[i][j] = A[i + newSize][j + newSize];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + newSize];
            B21[i][j] = B[i + newSize][j];
            B22[i][j] = B[i + newSize][j + newSize];
        }
    }

    matrix M1 = strassen(add(A11, A22), add(B11, B22)); matrix
    M2 = strassen(add(A21, A22), B11); matrix M3 = strassen(A11,
    subtract(B12, B22)); matrix M4 = strassen(A22, subtract(B21,
    B11)); matrix M5 = strassen(add(A11, A12), B22); matrix M6 =
    strassen(subtract(A21, A11), add(B11, B12));
    matrix M7 = strassen(subtract(A12, A22), add(B21, B22));

    matrix C11 = add(subtract(add(M1, M4), M5), M7);
```

```
        matrix C12 = add(M3, M5); matrix C21 =
        add(M2, M4);
        matrix C22 = add(subtract(add(M1, M3), M2), M6);


        matrix C(n, vector<int>(n));
        for (int i = 0; i < newSize; ++i) { for
            (int j = 0; j < newSize; ++j) {
                C[i][j] = C11[i][j];
                C[i][j + newSize] = C12[i][j];
                C[i + newSize][j] = C21[i][j];
                C[i + newSize][j + newSize] = C22[i][j];

            }
        }

return C;
    }


    void printMatrix(const matrix &M) { for
        (const auto &row : M) {
            for (int val : row) { cout
                << val << " ";
            }
            cout << endl;
        }
    }

 int main() { int n = 4; matrix A = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13,
     14, 15, 16}}; matrix B = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14,
    15, 16}};
```

```
        matrix C = strassen(A, B);

 cout << "Result matrix:" << endl;
        printMatrix(C);

        return 0;
    }
```



```
Result matrix:
52 24 17 38
72 53 24 73
56 38 23 31
59 42 21 73
```

Experiment-7

1.Aim

To implement a Heap data structure (Max or Min Heap), use it to perform Heap-Sort, and build a Priority Queue.

2.Objective

1.Understand the properties and operations of a Heap.
2.Implement a Max Heap or Min Heap.
3.Use the Heap to perform Heap-Sort. 4.Utilize the Heap to construct a Priority Queue.

3.Algorithm

1. Implementing a Heap (Max or Min Heap) Max Heap:

   1. Insert Element:
      - Add the element to the end of the heap.
      - Compare the added element with its parent; if the added element is greater, swap them.
      - Repeat the process until the heap property is restored.
2. Delete Element:
      - Remove the root element.
      - Replace the root with the last element in the heap.

    - Compare the new root with its children; if the new root is smaller, swap it with the larger child.
    - Repeat the process until the heap property is restored.

Min Heap:

1. Insert Element:
    - Add the element to the end of the heap.
    - Compare the added element with its parent; if the added element is smaller, swap them.
    - Repeat the process until the heap property is restored.
2. Delete Element:
    - Remove the root element.
    - Replace the root with the last element in the heap.
    - Compare the new root with its children; if the new root is larger, swap it with the smaller child.
    - Repeat the process until the heap property is restored.

2. Heap-Sort Algorithm

1. Build a Max Heap from the input data.
2. Extract the maximum element (root of the heap) and swap it with the last element of the heap.
3. Reduce the heap size by one and heapify the root element to restore the heap property.
4. Repeat the process until all elements are sorted.

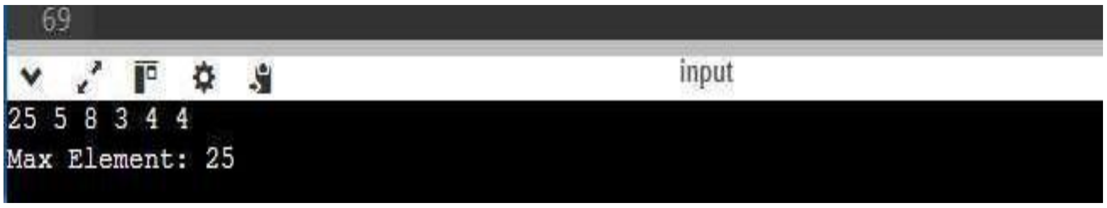
3. Priority Queue using Heap

1. Insert Element:
    - Add the element to the heap.
    - Restore the heap property by comparing the added element with its parent and swapping if necessary.
2. Extract Maximum/Minimum:
    - Remove the root element (highest priority).
    - Replace the root with the last element in the heap.
    - Restore the heap property by comparing the new root with its children and swapping if necessary

4. Implementation

1. heap

```
#include      <iostream>

#include <vector> using

namespace

std; class MaxHeap {

private:

    vector<int> heap;  void heapifyUp(int index) { int

    parentIndex = (index - 1) / 2; if (index > 0&&

    heap[index]        >        heap[parentIndex])        {

    swap(heap[index],                heap[parentIndex]);

    heapifyUp(parentIndex);  } }

   void heapifyDown(int index) { int leftChild = 2 * index +

       1; int rightChild = 2 * index + 2; int largest = index; if

       (leftChild < heap.size() && heap[leftChild] >

       heap[largest]) { largest = leftChild; } if (rightChild <

       heap.size() && heap[rightChild] > heap[largest]) {

       largest = rightChild;        }

       if (largest != index) {

           swap(heap[index], heap[largest]);

           heapifyDown(largest);        } }

public:

    void insert(int element) {

       heap.push_back(element);

       heapifyUp(heap.size() - 1); } int

    extractMax() { if (heap.size() == 0)

    return    1; int maxElement =

    heap[0]; heap[0] = heap.back();

    heap.pop_back();

    heapifyDown(0);                 return

    maxElement; } void printHeap() {

    for (int i : heap) {

           cout << i << " ";          }

cout << endl; }}; int main() {
   MaxHeap maxHeap;

   maxHeap.insert(5)

    ;

   maxHeap.insert(25); maxHeap.insert(4);

   maxHeap.insert(3); maxHeap.insert(4);

   maxHeap.insert(8); maxHeap.printHeap

   ();

    cout << "Max Element: " << maxHeap.extractMax() <<

    endl; return 0;

}
```

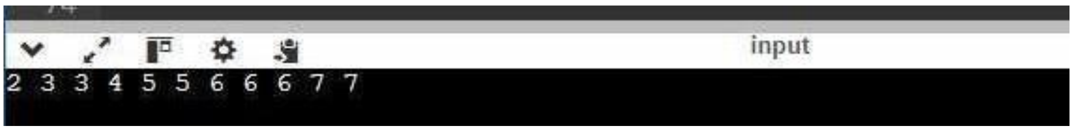


```
25 5 8 3 4 4
Max Element: 25
```

2. heap implement the Heap-Sort algorithm.

```
#include       <iostream>

#include <vector> using

namespace

std; class MaxHeap {

private:

    vector<int> heap; void heapifyUp(int

    index) { int parentIndex = (index - 1) /

    2; if (index > 0 && heap[index] >

    heap[parentIndex])                          {

    swap(heap[index],

    heap[parentIndex]);


           heapifyUp(parentIndex);         } }

   void heapifyDown(int index) { int leftChild = 2 * index +

      1; int rightChild = 2 * index + 2; int largest = index; if

      (leftChild < heap.size() && heap[leftChild] >

      heap[largest]) { largest = leftChild; } if (rightChild <
```

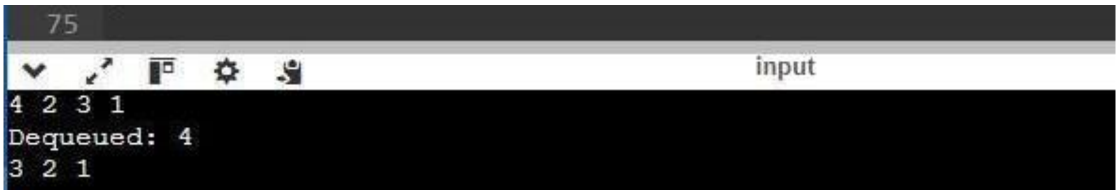
```
        heap.size() && heap[rightChild] > heap[largest]) {
        largest = rightChild;       }
        if (largest != index) {
            swap(heap[index], heap[largest]);
            heapifyDown(largest);       } }
public:
    void insert(int element) {
        heap.push_back(element);
        heapifyUp(heap.size()    -    1);    }    int
    extractMax() { if (heap.size() == 0) return 1;
    int  maxElement  =  heap[0];  heap[0]  =
    heap.back();                 heap.pop_back();
    heapifyDown(0); return maxElement; } void
    buildHeap(vector<int>&  arr)  {  heap  =  arr;
    for (int i = heap.size() / 2 - 1; i >= 0; i-
           -)  {  heapifyDown(i);  }  }
    void heapSort(vector<int>&
        arr) { buildHeap(arr);
        for (int i = arr.size() - 1; i >= 0; i--) {

             arr[i] = extractMax();        } }
};

int main() { vector<int> arr = {3, 1, 4, 1, 5, 9, 2,
    6, 5, 3,
    5}; MaxHeap  maxHeap; maxHeap.heapSort(arr);
    for (int i : arr) {
         cout << i << " "; }
    cout << endl; return 0;
}
```



```
2 3 3 4 5 5 6 6 6 7 7
```

3. heap to build a Priority Queue.

```
#include      <iostream>
#include <vector> using
namespace
std; class MaxHeap {
private:
    vector<int> heap; void heapifyUp(int
    index) { int parentIndex = (index - 1) /
    2; if (index > 0 && heap[index] >
    heapifyUp(parentIndex);

       }
    }
    void heapifyDown(int index) { int leftChild = 2 * index +
       1; int rightChild = 2 * index + 2; int largest = index; if
       (leftChild   <   heap.size()   &&   heap[leftChild]   >
       heap[largest]) { largest = leftChild; } if (rightChild <
       heap.size() && heap[rightChild] > heap[largest]) {
       largest = rightChild; } if (largest != index) {
          swap(heap[index], heap[largest]);
          heapifyDown(largest);}} public:
    void insert(int element) {
       heap.push_back(element);
       heapifyUp(heap.size() - 1); } int
    extractMax() { if (heap.size() == 0)
    return    1; int maxElement =
    heap[0]; heap[0] = heap.back();
    heap.pop_back();
    heapifyDown(0);              return
    maxElement;
    }
    void printHeap() {
       for (int i : heap) { cout <<
          i << " ";
       }cout << endl; } }; class PriorityQueue { private:
    MaxHeap          maxHeap;
public:
    void      enqueue(int      element)      {
       maxHeap.insert(element); } int
    dequeue() {
       return maxHeap.extractMax(); } void
    printQueue() {
       maxHeap.printHeap(); } }; int
main() { PriorityQueue pq; pq.enqueue(4);
    pq.enqueue(1); pq.enqueue(3); pq.enqueue(2);
    pq.printQueue(); cout << "Dequeued: " <<
    pq.dequeue() << endl; pq.printQueue(); return 0;
    }
```



```
4 2 3 1
Dequeued: 4
3 2 1
```

Experiment-8

1. Aim

To maximize the total value of items placed in a knapsack with a given capacity, allowing for the inclusion of fractional parts of items and Minimum Spanning Tree (MST) for a weighted connected graph, ensuring that all vertices are connected with the minimum possible total edge weight.

2. Objective

1. Determine the value-to-weight ratio of each item.
2. Sort items based on their value-to-weight ratios.
3. Select items to fill the knapsack in a greedy manner, taking whole items first and then fractions if necessary
4. Construct a spanning tree that includes all vertices.
5. Ensure that the total weight of the edges in the spanning tree is minimized.
6. Use a greedy approach to build the MST step-by-step.

3. Algorithm

1. Fractional Knapsack Problem

Calculate the value-to-weight ratio for each item.

Sort the items in descending order based on the ratio.
Initialize total value to 0 and iterate through the sorted items:

- If the item can fit entirely in the knapsack, add its full value and reduce the remaining capacity.
- If the item cannot fit, take the fraction that fits and add the corresponding value.

Return the total value accumulated in the knapsack.

2. Prim's Algorithm for Minimum Spanning Tree

Initialize a key array to store the minimum weight edge for each vertex and a parent array to reconstruct the MST.

Set the key of the first vertex to 0 and all others to infinity.

Mark all vertices as not included in the MST.

For V-1 iterations (where V is the number of vertices):

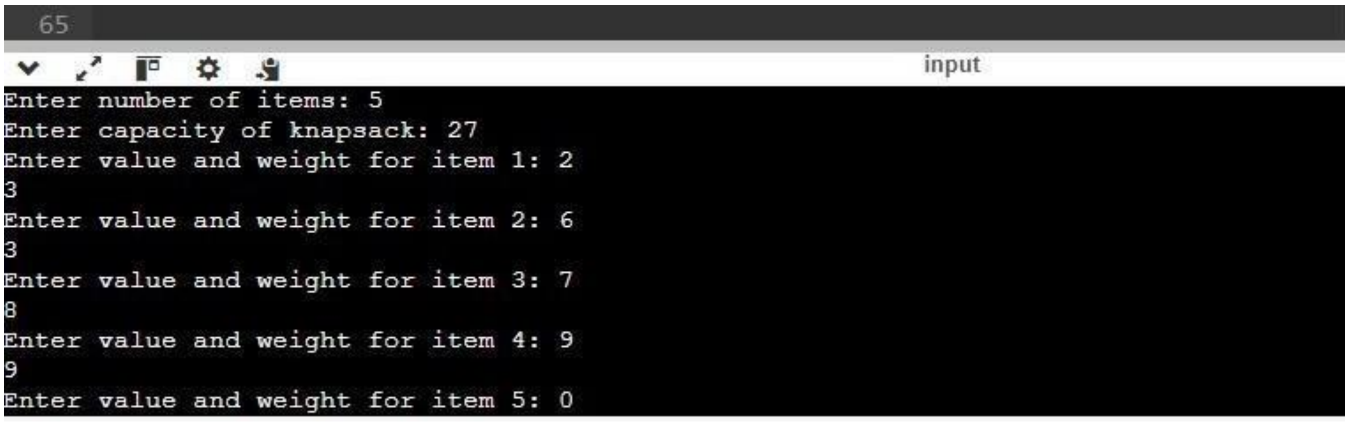
- Select the vertex with the minimum key that has not yet been included in the MST.
    - Mark this vertex as included.
    - Update the key and parent for adjacent vertices if a smaller edge weight is found.

After constructing the MST, print the edges along with their weights

4. Implementation

1. Fractional Knapsack Problem

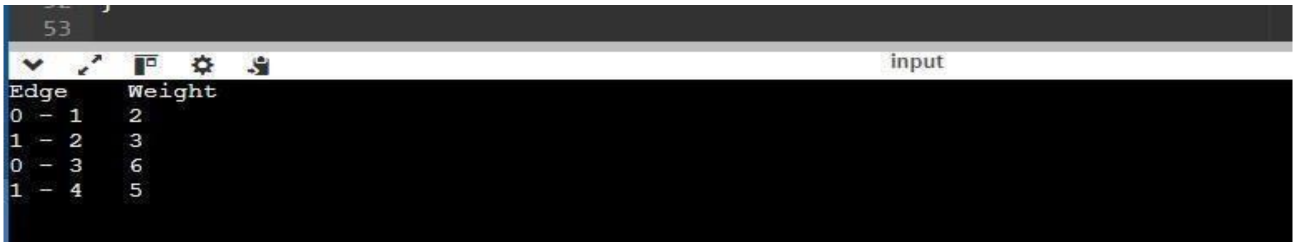
```
#include <iostream>
#include <vector>
#include
<algorithm> using
namespace std;
struct Item { int
value; int weight;
double ratio;
};
bool compare(Item a, Item b) {
return a.ratio > b.ratio;
double fractionalKnapsack(int capacity, vector<Item> items) { for
(auto &item : items) {
item.ratio = (double)item.value / item.weight;
}
sort(items.begin(), items.end(),
compare); double totalValue = 0.0; for
(auto &item : items) { if (capacity <= 0)
break; if (item.weight <= capacity) {
totalValue += item.value;
capacity -= item.weight;
} else {
totalValue    +=    item.ratio    *    capacity;
capacity = 0;
}
}
return totalValue;
}
int main() { int n, capacity; cout
<< "Enter number of items: "; cin
>> n;
cout << "Enter capacity of knapsack:
"; cin >> capacity; vector<Item>
items(n); for (int i = 0; i < n; i++) {
cout << "Enter value and weight for item " << i + 1 << ": "; cin
>> items[i].value >> items[i].weight;
}
0;


double maxValue = fractionalKnapsack(capacity, items);
cout       <<       "Maximum       value       in       the
knapsack;"<<maxValue<<endl;return 0;
```



```
Enter number of items: 5
Enter capacity of knapsack: 27
Enter value and weight for item 1: 2
3
Enter value and weight for item 2: 6
3
Enter value and weight for item 3: 7
8
Enter value and weight for item 4: 9
9
Enter value and weight for item 5: 0
```

2. Prim's Algorithm for Minimum Spanning Tree

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
const int V = 5;
int minKey(int key[], bool mstSet[]) {
int min = INT_MAX, min_index;
for (int v = 0; v < V; v++) { if
(!mstSet[v] && key[v] < min) {
min = key[v];
min_index = v;
}
}
return min_index;
}
void primMST(int graph[V][V]) {
int      parent[V];
int key[V];
bool mstSet[V];
for (int i = 0; i < V; i++) {
key[i] = INT_MAX;
mstSet[i] = false;
}
key[0]         =         0;
parent[0] = -1;
for (int count = 0; count < V - 1; count++) {
int u = minKey(key,
mstSet); mstSet[u] = true;
for (int v = 0; v < V; v++) {
if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])  {
parent[v] = u;
key[v] = graph[u][v];
}
}
}
cout << "Edge \tWeight\n";
for (int i = 1; i < V; i++) {
cout << parent[i] << "
-
" << i << "\t" << graph[i][parent[i]] << " \n";
}
}
int main() { int
graph[V][V] = {
{0, 2, 0, 6, 0},
{2, 0, 3, 8, 5},
{0, 3, 0, 0, 7},
{6, 8, 0, 0, 9},
}
{0, 5, 7, 9, 0}
};
primMST(graph);

return 0;

}
```



```
Edge     Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

Experiment-9

1. Aim

To efficiently manage and query the connected components of a graph and to find the Minimum Spanning Tree (MST) for a given weighted connected graph.

2. Objective

1. Implementing the Union-Find algorithm to detect cycles in an undirected graph.
2. Utilizing the Union-Find data structure to execute Kruskal's algorithm, ensuring efficient cycle detection during the edge selection process

3. Algorithm

Initialize: Create parent and rank arrays for all vertices.
Find Operation: find(x):
If x is its own parent, return x.
Otherwise, recursively find the root parent of x and apply path compression.
Union Operation: union(x, y):
Find roots of x and y.
Attach the tree with a lower rank to the root of the tree with a higher rank. If
ranks are the same, attach one to another and increment the rank.

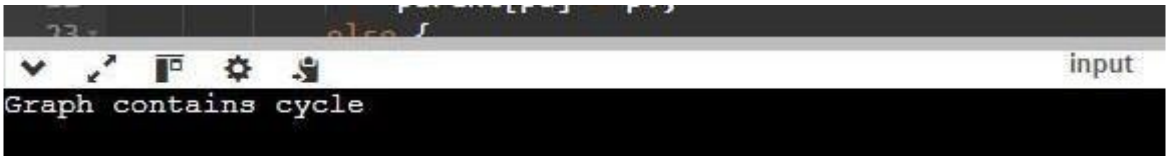
2.Kruskal's algorithm for finding the Minimum Spanning Tree

Sort Edges: Sort all edges of the graph in non-decreasing order of their weight.
Initialize: Create a disjoint-set for all vertices.
Edge Selection:
For each edge (u, v):
If find(u) != find(v): This ensures no cycle is formed.

Add the edge to the MST.
Perform union(u, v).

4.Implementation 1. Disjoint Set Data Structure (Union-Find algorithm)

```
#include <iostream>
#include <vector>
using namespace std;
class DisjointSet {
public:
DisjointSet(int n) : parent(n), rank(n, 0) {
for (int i = 0; i < n; ++i)
parent[i] = i;
}
int find(int u) { if (u !=
parent[u]) parent[u] =
find(parent[u]);
return parent[u];
}
void unionSets(int u, int v)
{ int pu = find(u); int pv =
find(v); if (pu != pv) { if
(rank[pu] > rank[pv])
parent[pv] = pu; else if
(rank[pu] < rank[pv])
parent[pu] = pv; else {
parent[pv] = pu;
rank[pu]++;
}
}
}
private:
vector<int> parent;
vector<int> rank;
};
bool hasCycle(int V, vector<pair<int, int>>& edges)
{ DisjointSet ds(V); for (auto& edge : edges) { int u =
edge.first; int v = edge.second; if (ds.find(u) ==
ds.find(v)) return true;
ds.unionSets(u, v);
}
return false;
}
int main() {
int V = 5;
vector<pair<int, int>> edges = { {0, 1}, {1, 2}, {2, 3}, {3, 4}, {4, 1}
};
if (hasCycle(V, edges)) cout << "Graph
contains cycle" << endl;
        else
cout << "Graph does not  contains cycle" << endl;

return 0;

}
```



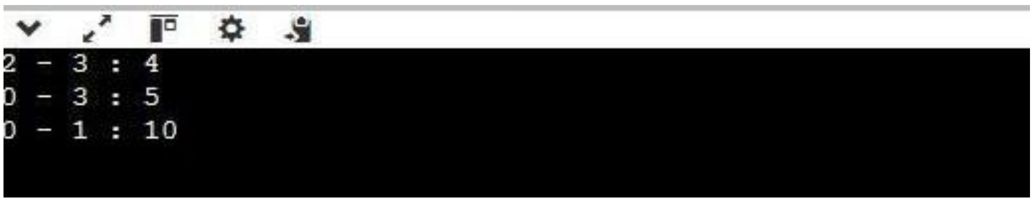
```
Graph contains cycle
```

2. Kruskal's algorithm for finding the Minimum Spanning Tree

```
#include

<iostream>

#include <vector>

#include

<algorithm>

using namespace

std;

class      DisjointSet      {

public:

   DisjointSet(int n) : parent(n), rank(n, 0)

       {  for  (int  i  =  0;  i  <  n;  ++i)

           parent[i] = i;

   }

   int find(int u) {  if

       (u !=  parent[u])

       parent[u]         =

       find(parent[u]);

       return

       parent[u];
```

```
  }
  void unionSets(int u, int
    v)   {   int   pu   =
    find(u); int pv =
    find(v); if (pu
    != pv) { if (rank[pu]
      >          rank[pv])
      parent[pv] = pu;

      else   if   (rank[pu]   <
      rank[pv])  parent[pu]  =
      pv; else {
        parent[pv] =
        pu;
        rank[pu]++;
      }
    }
  }
private:
  vector<int>
  parent;
  vector<int>
  rank;
};
struct Edge { int
  u, v, weight;
  bool     operator<(const     Edge&
    other)  const  {  return  weight  <
    other.weight;
  }
};
```

```
vector<Edge> kruskalMST(int V, vector<Edge>&
   edges)        {        DisjointSet        ds(V);
   vector<Edge> mst;
   sort(edges.begin(),
   edges.end()); for (auto& edge :
   edges)  {  if  (ds.find(edge.u)  !=
      ds.find(edge.v))                    {
      ds.unionSets(edge.u,   edge.v);
      mst.push_back(edge);
      }
   }
   return mst;
}
int main() {int
   V     =     4;
   vector<Edg
   e> edges =
   { {0, 1, 10},
   {0, 2, 6}, {0,
   3, 5}, {1, 3,
   15},  {2,  3,
   4} };
   vector<Edge> mst = kruskalMST(V, edges); cout << "Edges in
   the Minimum Spanning Tree:" << endl; for (auto& edge : mst)
   { cout << edge.u << " - " << edge.v << " : " << edge.weight <<
   endl;
   }
   return 0;
}
```



```
2 - 3 : 4
0 - 3 : 5
0 - 1 : 10
```

Experiment-10

Dijkstra's algorithm

1. Aim

To implement Dijkstra's Algorithm to find the shortest path from a specific source vertex to all other vertices in a weighted connected graph. The graph will be represented in two different formats: a **Weighted Matrix** and an **Adjacency List**..

2. Objective

1. **Understand Dijkstra's Algorithm**:

- Review the theory behind Dijkstra's Algorithm, its mechanics, and how it guarantees finding the shortest path in a weighted, connected graph.

2. **Implement Dijkstra's Algorithm**:

- Develop a working solution for Dijkstra's Algorithm that finds the shortest path from a single source vertex to all other vertices in a graph.

2. **Graph Representation**:

- **Weighted Matrix Representation**: Implement the algorithm using a 2D matrix where each entry represents the weight of the edge between two vertices. If no edge exists, the weight can be set to infinity.
- **Adjacency List Representation**: Implement the algorithm using a list where each vertex has a list of its neighboring vertices and the corresponding edge weights.

2. **Compare the Two Representations**:

- Compare the performance and ease of implementation of Dijkstra's algorithm using both the **Weighted Matrix** and the **Adjacency List** representations.

2. **Evaluate the Algorithm**:

- Evaluate the time complexity, memory usage, and overall efficiency of the algorithm in both graph representations.

3. Algorithm

1. weighted matrix representation

Step1-Initialize dist[] as described above.

Step2-Initialize the priority queue with pairs (dist[u],u)(dist[u], u)(dist[u],u).

Step3-Extract the vertex with the minimum distance dist[u]dist[u]dist[u] from the priority queue.

Step4-For each vertex v, check if there is an edge between u and v (i.e., W[u][v]<∞W[u][v] < \inftyW[u][v]<∞).

- If so, update the distance dist[v] if a shorter path is found.

Step5-Continue until the priority queue is empty

2. Adjacency List Representation

Initialize Data Structures:

- Create an array dist[] of size V to store the shortest distance from the source to each vertex. Initialize all elements to infinity (inf), except for the source vertex, which is initialized to

0. ● Create a boolean array visited[] to track which vertices have been processed.

Priority Queue:

- Use a priority queue (min-heap) to store the vertices and their current shortest distance, where the priority is the smallest distance.

Main Loop:

- Extract the vertex u with the smallest tentative distance from the priority queue.
- For each neighbor v of vertex u (which is obtained from the adjacency list), calculate the tentative distance dist[u] + weight(u, v). If this distance is smaller than the current distance to v, update dist[v] and add v to the priority queue with the new distance.

End Condition:

- Repeat the process until all vertices have been processed or the priority queue is empty.

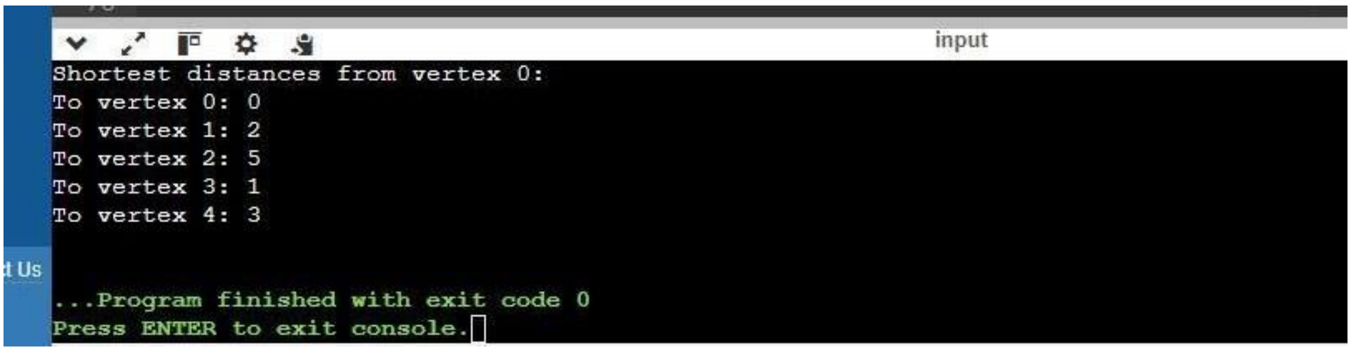
4.Implementation

1. Represent the graph as Weighted

Matrix

```
(Dijkstra's

algorithm)            #include

<iostream>

#include        <climits>

#include <vector>

using namespace

std;

int  minDistance(const  vector<int>&
dist, const vector<bool>& sptSet, int
V) {

   int       min       =

   INT_MAX; int

   minIndex = -1;

for (int v = 0; v < V; v++) {

      if (!sptSet[v] && dist[v] <=

      min) { min = dist[v];
```

```
        minIndex = v;


        }


    }
    return minIndex;


}


vector<int>     dijkstra(const

vector<vector<int>>&

graph, int source) {


    int V = graph.size();


    vector<int> dist(V,


    INT_MAX);




    dist[source] = 0;


    vector<bool> sptSet(V,


    false);


    for (int count = 0; count < V - 1; count++)

{


        int u = minDistance(dist,


   sptSet, V); sptSet[u] = true;


        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] !=
0 && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {

        dist[v] = dist[u] + graph[u][v];

            }

        }

    }

    return dist;

}

int main() {

    vector<vector<int>>

    graph = {

        {0, 2, 0, 1, 0},

        {2, 0, 3, 0, 0},

        {0, 3, 0, 4, 6},

        {1, 0, 4, 0, 2},
        {0, 0, 6, 2, 0}

    };
    int source = 0;

    vector<int>
shortest_distances
=       dijkstra(graph,
source);

    cout << "Shortest distances from vertex "
<< source << ":\n";

    for      (int      i      =      0;      i      <
shortest_distances.size(); i++) {

    cout << "To vertex " << i << ":
" << shortest_distances[i] << endl;

    }

    return 0;

}
```

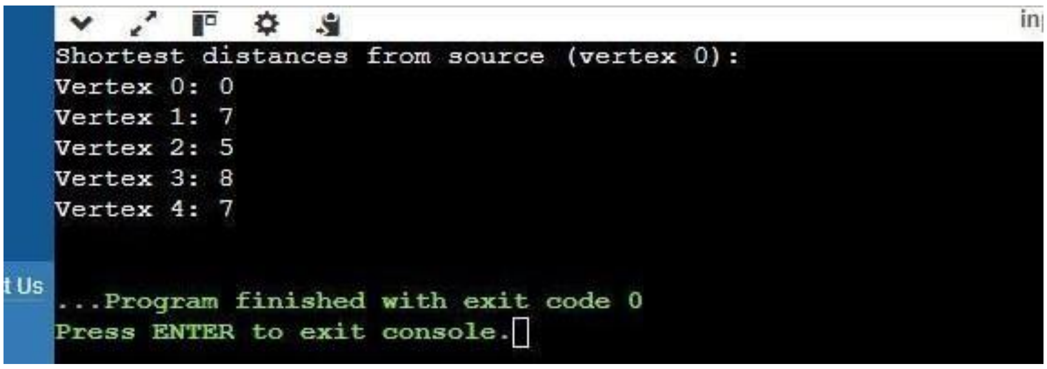


```
Shortest distances from vertex 0:
To vertex 0: 0
To vertex 1: 2
To vertex 2: 5
To vertex 3: 1
To vertex 4: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

2. Represent the graph as Adjacency List

```
#include
<iostream>
```

```
#include        <vector>
#include <queue>
#include <climits>
#include
<functional>

using    namespace    std;

typedef pair<int, int>

pii;


class
Graph
{
public:
int V;
  vector<vector<pii>> adjList;

  Graph(int V)
      { this->V =
      V;
      adjList.resize(V);
  }

   void addEdge(int u, int v, int
      weight) {
      adjList[u].push_back({v,
      weight});
      adjList[v].push_back({u,
      weight});
  }

  vector<int>     dijkstra(int
     source) {  vector<int>
     dist(V, INT_MAX);
     dist[source] = 0;


      priority_queue<pii, vector<pii>,
      greater<pii>> pq; pq.push({0, source});

      while
        (!pq.empty()) {
        int       u       =
        pq.top().secon
        d;    int    d    =
        pq.top().first;
        pq.pop();


        if     (d     >
           dist[u]) {
           continue
           ;
        }
    for (auto& edge :
           adjList[u])    {
           int      v      =
           edge.first;
           int weight = edge.second;


        if (dist[u] + weight <
              dist[v]) { dist[v] =
              dist[u]   +   weight;
              pq.push({dist[v],
              v});
           }
        }
      }

      return dist;
   }
 };

int  main()   {

   Graph

  g(5);


  g.addEdge(0, 1, 10);
  g.addEdge(0, 2, 5);
  g.addEdge(1, 2, 2);
  g.addEdge(1, 3, 1);
  g.addEdge(2, 3, 9);
  g.addEdge(2, 4, 2);
  g.addEdge(3, 4, 4);


  vector<int> dist = g.dijkstra(0);

  cout << "Shortest distances from source (vertex 0):"
  << endl; for (int i = 0; i < dist.size(); ++i) { if
     (dist[i] == INT_MAX) {
         cout << "Vertex " << i << " is unreachable." << endl;
      } else { cout << "Vertex " << i << ": " << dist[i] <<
         endl;
      }
  }

  return 0;
}
```



```
Shortest distances from source (vertex 0):
Vertex 0: 0
Vertex 1: 7
Vertex 2: 5
Vertex 3: 8
Vertex 4: 7


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment -11

Fibonacci

1. Aim:

To find the n-th Fibonacci number using the Dynamic Programming approach in C++. The approach will efficiently calculate Fibonacci numbers by storing previously calculated values to avoid redundant calculations, which significantly improves performance over the naive recursive method.

2. Objective:

The primary objective of this program is to:

1. Implement the Fibonacci sequence using Dynamic Programming to minimize time complexity.
2. Store intermediate Fibonacci values in a table (array) to reuse them, avoiding redundant calculations.

3. Output the n-th Fibonacci number.

3.Algorithm:

1. Initialization:

○ Create an array dp[] where dp[i] will store the i-th Fibonacci number.

○ Initialize the first two Fibonacci numbers:

■ dp[0] = 0 (Fibonacci number 0)

■ dp[1] = 1 (Fibonacci number 1)

2. Iterative Calculation:

○ For each i from 2 to n, calculate the i-th Fibonacci number using the recurrence relation: dp[i]=dp[i−1]+dp[i−2]dp[i] = dp[i-1] + dp[i-2]dp[i]=dp[i−1]+dp[i−2]

○ Store the result of each calculation in the array dp[].

3. Return the n-th Fibonacci number:

○ After the loop, the value dp[n] will hold the n-th Fibonacci number.

4. Edge Case:

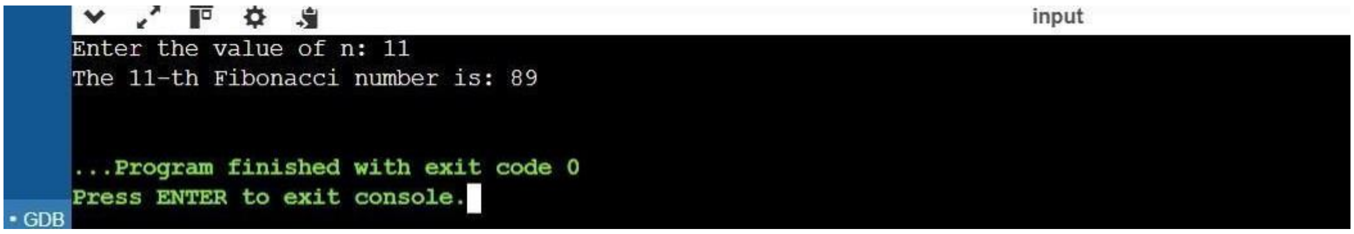
○ If n is 0, return 0.

○ If n is 1, return 1

4.IMPLEMENTATION:-

```
#include <iostream>
#include <vector>
using namespace
std; int fibonacci(int
n) { if (n <= 1) return
n; vector<int> dp(n +
1); dp[0] = 0; dp[1] =
1;
for (int i = 2; i <= n; i++) { dp[i]
= dp[i - 1] + dp[i - 2];
}
return dp[n];
}
int main() {
int n;
cout << "Enter a number: ";
cin >> n;
cout << "The " << n << "-th Fibonacci number is: " << fibonacci(n) << endl; return
0;
```

}



```
Enter the value of n: 11
The 11-th Fibonacci number is: 89


...Program finished with exit code 0
Press ENTER to exit console.
```

LCS

1. Aim:

To implement the Longest Common Subsequence (LCS) algorithm using the Dynamic Programming approach in C++. The goal is to find both the length of the LCS and the actual subsequence(s). If multiple LCS solutions are possible, the algorithm should find and print all such subsequences.

2. Objective:

The main objectives of this program are:

1. To implement the LCS algorithm using Dynamic Programming to efficiently compute the longest common subsequence between two strings.
2. To determine the length of the LCS.
3. To reconstruct and print all possible LCSs if multiple solutions exist.

3.Algorithm:

The algorithm for solving the LCS problem using Dynamic Programming can be broken down as follows:

1. Initialization:

o Given two strings X of length m and Y of length n, create a 2D table dp[m+1][n+1] where dp[i][j] will store the length of the LCS of the first i characters of string X and the first j characters of string Y.

2. Filling the DP Table:

o If X[i-1] == Y[j-1], then dp[i][j] = dp[i-1][j-1] + 1 (i.e., the current characters are part of the LCS, so we add 1 to the length from the previous characters).

o If X[i-1] != Y[j-1], then dp[i][j] = max(dp[i-1][j], dp[i][j-1]) (i.e., we take the maximum length from either excluding the current character from X or from Y).

3. Length of the LCS:

○ The length of the LCS will be stored in dp[m][n], which represents the length of the longest common subsequence between the two strings.

4. Reconstructing the LCS:

○ Starting from dp[m][n], backtrack to find the actual subsequence(s). If multiple subsequences are possible, follow both possible paths when characters match and store the LCSs.

5. Handling Multiple LCSs:

○ If multiple LCSs are possible, use backtracking to generate all subsequences. Every time we encounter a match, we follow both possible paths (i.e., move up or left in the DP table).

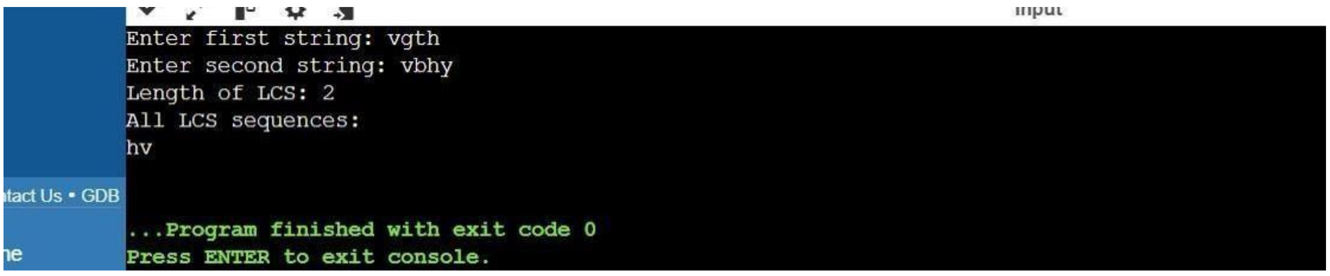
6. Edge Cases:

○ If one or both strings are empty, the LCS length is 0, and the LCS is an empty string.

4.IMPLEMENTATION:-

```
#include <iostream>
#include <vector>
#include <string>
#include <set> #include <algorithm> using namespace std; int
LCSLength(const string& X, const string& Y, vector<vector<int>>& dp) { int
m = X.size(); int n = Y.size(); for (int i = 0; i <= m; ++i) { for (int j = 0; j <= n;
++j) { if (i == 0 || j == 0) { dp[i][j] = 0; // base case } else if (X[i-1] == Y[j-1])
{ dp[i][j] = dp[i-1][j-1] + 1;
} else { dp[i][j] = max(dp[i-1][j],
dp[i][j-1]);
}
}
}
return dp[m][n];
}
void findLCS(int i, int j, const string& X, const string& Y, const vector<vector<int>>& dp, string&
currentLCS, set<string>& result) { if (i == 0 || j == 0) { result.insert(currentLCS); // Reached the top-
left, insert LCS return;
} if (X[i - 1] == Y[j - 1]) {
currentLCS.push_back(X[i - 1]); findLCS(i - 1, j
- 1, X, Y, dp, currentLCS, result);
currentLCS.pop_back(); // Backtrack
} else {
if (dp[i - 1][j] == dp[i][j]) { findLCS(i - 1, j, X,
Y, dp, currentLCS, result);
} if (dp[i][j - 1] == dp[i][j]) { findLCS(i, j - 1,
X, Y, dp, currentLCS, result);
}
}
}
int main() { string X, Y; cout <<
"Enter first string: "; cin >> X;
cout << "Enter second string: ";
cin >> Y; int m = X.size(); int n =
Y.size();
// Create DP table for storing lengths of LCS
vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
// Compute the length of the LCS int lcsLength
= LCSLength(X, Y, dp); cout << "Length of LCS:
" << lcsLength << endl; // Set to store all LCS
(to avoid duplicates) set<string> result;
// String to store the current LCS during backtracking
string currentLCS; // Backtrack to find all LCS
findLCS(m, n, X, Y, dp, currentLCS, result); cout <<
"All LCS sequences:" << endl; for (const string& lcs :
result) { cout << lcs << endl;
}
return 0;


}
```

Output –



```
Enter first string: vgth
Enter second string: vbhy
Length of LCS: 2
All LCS sequences:
hv


...Program finished with exit code 0
Press ENTER to exit console.
```