मौलाना आज़ाद राष्ट्रीय प्रौद्योगिकी संस्थान, भोपाल – 462003
MAULANA AZAD NATIONAL INSTITUTE OF TECHNOLOGY BHOPAL – 462003

कंप्यूटर विज्ञान एवं अभियांत्रिकी विभाग
(Department of Computer Science and Engineering)

प्रयोग सूची (Lab Manual)

सीएसई- द्वितीय वर्ष /चतुर्थ सेमेस्टर (CSE- II Year / IV Semester)

सत्र:जनवरी-जून, २०२५ (Session: Jan-April, 2025)

एल्गोरिदम का विश्लेषण और डिजाइन (सीएसई-226) / ADA Lab (CSE-226)

Lab: 01

1. अपनी पसंद की प्रोग्रामिंग लैंग्वेज में एक प्रोग्राम लिखें, जो दिए गए पॉज़िटिव इंटीजर 'N' तक के सभी प्राइम नंबर्स को सिव ऑफ एरैटोस्थनीज एल्गोरिदम का उपयोग करके खोजे। इसके अतिरिक्त, अपने इंप्लीमेंटेशन की टाइम कॉम्प्लेक्सिटी का विश्लेषण करें और प्रदान करें। यह समझाएं कि टाइम कॉम्प्लेक्सिटी कैसे प्राप्त की गई है और यह इनपुट साइज 'N' पर कैसे निर्भर करती है।

Write a program in a programming language of your choice to find all prime numbers up to a given positive integer 'N' using the Sieve of Eratosthenes algorithm. Additionally, analyze and provide the time complexity of your implementation. Explain how the time complexity is derived and how it depends on the input size 'N'.

ALGORITHM Sieve(n)
INPUT: A positive integer $n \geq 2$
OUTPUT: Array L of all prime numbers less than or equal to n

```
// Step 1: Initialize the array A
LET A[2..n] ← each element A[p] initialized to p

// Step 2: Mark non-prime numbers
FOR p ← 2 TO ⌊√n⌋ DO
  IF A[p] ≠ 0 THEN              // If p is not marked, it's a prime
    LET j ← p × p
    WHILE j ≤ n DO
      A[j] ← 0                  // Mark multiple of p as non-prime
      j ← j + p
    END WHILE
  END IF
END FOR

// Step 3: Collect remaining prime numbers
LET L[0..] ← empty list
LET i ← 0
```

```
FOR p ← 2 TO n DO
  IF A[p] ≠ 0 THEN
    L[i] ← A[p]            // Store the prime number
    i ← i + 1
  END IF
END FOR

// Step 4: Return the list of primes
RETURN L
END ALGORITHM
```

Sample Input

- **n = 30**

Expected Output

- **L = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]**

Time Complexity: **O(n · ln(ln(n)))**

---

2. एक प्रोग्राम लिखें जो यह निर्धारित करे कि दी गई स्क्वेयर मैट्रिक्स एक मैजिक स्क्वेयर है या नहीं। एक मैजिक स्क्वेयर को एक $n \times n$ मैट्रिक्स के रूप में परिभाषित किया जाता है, जिसमें 1 से n^2 तक के विशिष्ट तत्व होते हैं, जहां प्रत्येक रो, कॉलम और डायगोनल का योग समान संख्या के बराबर होता है।

Write a program to determine whether a given square matrix is a Magic Square or not. A Magic Square is defined as an $n \times n$ matrix of distinct elements ranging from 1 to n^2, where the sum of every row, column, and diagonal is equal to the same number.

ALGORITHM isMagicSquare(mat)
INPUT: A 3x3 matrix mat
OUTPUT: Boolean value (True if mat is a Magic Square, False otherwise)

```
// Step 1: Set matrix size
LET n ← 3  // Matrix is fixed at 3x3

// Step 2: Calculate the sums of the two diagonals
LET sumd1 ← 0       // Sum of the main diagonal
LET sumd2 ← 0       // Sum of the secondary diagonal

FOR i ← 0 TO n - 1 DO
  sumd1 ← sumd1 + mat[i][i]          // Main diagonal
  sumd2 ← sumd2 + mat[i][n - 1 - i]  // Secondary diagonal
END FOR

// Step 3: Check if both diagonal sums are equal
```

```
IF sumd1 ≠ sumd2 THEN
   RETURN False
END IF

// Step 4: Check if all rows and columns sum to the same value
FOR i ← 0 TO n - 1 DO
   LET rowSum ← 0
   LET colSum ← 0

   FOR j ← 0 TO n - 1 DO
      rowSum ← rowSum + mat[i][j]
      colSum ← colSum + mat[j][i]
   END FOR

   IF rowSum ≠ colSum OR colSum ≠ sumd1 THEN
      RETURN False
   END IF
END FOR

// Step 5: If all checks passed, it's a magic square
RETURN True
END ALGORITHM
```

Sample Input

mat = [[8, 1, 6], [3, 5, 7], [4, 9, 2]]

Expected Output

True // The matrix is a magic square

Time Complexity Analysis: O(n^2)

Lab: 02

1. एक रिकर्सिव प्रोग्राम लिखें जो टावर ऑफ हनोई प्रॉब्लम को हल करे। यह प्रोग्राम सभी डिस्क्स को सोर्स रॉड से डेस्टिनेशन रॉड पर ऑक्सिलरी रॉड का उपयोग करके स्थानांतरित करे और कुल आवश्यक स्टेप्स की गणना करे।

   Write a recursive program to solve the Tower of Hanoi problem. The program should move all disks from the source rod to the destination rod using an auxiliary rod and count the total steps required.

   Here's the algorithm for solving the **Tower of Hanoi** problem using recursion:

   ALGORITHM Tower_Of_Hanoi(n, source, destination, auxiliary)
   INPUT:

- n: Number of disks
- source: Starting rod
- destination: Target rod
- auxiliary: Helper rod

OUTPUT:

- Step-by-step instructions to move all disks from source to destination
- Total number of steps required

```
// Global step counter
LET steps ← 0

// Step 1: Define the recursive function
FUNCTION tower_of_hanoi(n, source, destination, auxiliary)
   IF n == 1 THEN
      PRINT "Move disk 1 from", source, "to", destination
      steps ← steps + 1
      RETURN
   END IF

   // Step 2: Move n-1 disks from source to auxiliary
   CALL tower_of_hanoi(n - 1, source, auxiliary, destination)

   // Step 3: Move the nth disk from source to destination
   PRINT "Move disk", n, "from", source, "to", destination
   steps ← steps + 1

   // Step 4: Move n-1 disks from auxiliary to destination
   CALL tower_of_hanoi(n - 1, auxiliary, destination, source)
END FUNCTION

// Step 5: Initialize and invoke the function
LET n ← given number of disks
LET steps ← 0
CALL tower_of_hanoi(n, 'A', 'C', 'B')

// Step 6: Output the total number of steps
PRINT "Total steps required:", steps

END ALGORITHM
```

Sample Input
n = 3
Expected Output
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Total steps required: 7

Time Complexity Analysis: $O(2^n)$

---

2. एक प्रोग्राम लिखें जो रिकर्शन का उपयोग करके n-थ फिबोनाची नंबर की गणना करे। यह प्रोग्राम गणना के दौरान किए गए कुल रिकर्सिव कॉल्स की संख्या भी गिने।

Write a program to calculate the n-th Fibonacci number using recursion. The program should also count the total number of recursive calls made during the calculation.

ALGORITHM Fibonacci(n, call_count)
INPUT:

- n: Index of the Fibonacci number to compute
- call_count: A list with a single integer to track number of recursive calls

OUTPUT:

- The n-th Fibonacci number
- Total number of recursive calls made

```
// Step 1: Define the recursive function
FUNCTION fibonacci(n, call_count)
  // Step 2: Base Cases
  IF n == 0 THEN
    RETURN 0
  ELSE IF n == 1 THEN
    RETURN 1
  END IF

  // Step 3: Increment call count
  call_count[0] ← call_count[0] + 1

  // Step 4: Recursive computation
```

```
    RETURN fibonacci(n - 1, call_count) + fibonacci(n - 2, call_count)
END FUNCTION

// Step 5: Initialize counter and call function
LET n ← given index (e.g., 5)
LET call_count ← [0]        // Using list to simulate pass-by-reference
LET result ← fibonacci(n, call_count)

// Step 6: Output result
PRINT "The", n, "-th Fibonacci number is:", result
PRINT "Total recursive calls made:", call_count[0]

END ALGORITHM
```

Sample Input
n = 5

Expected Output
The 5-th Fibonacci number is: 5
Total recursive calls made: 15

Time Complexity Analysis: $O(2^n)$

---

3. एक प्रोग्राम लिखें जो रिकर्शन का उपयोग करके दो संख्याओं का GCD (ग्रेटेस्ट कॉमन डिवाइज़र) गणना करे। यह प्रोग्राम गणना के दौरान किए गए कुल रिकर्सिव कॉल्स की संख्या भी गिने।

Write a program to calculate the GCD (Greatest Common Divisor) of two numbers using recursion. The program should also count the total number of recursive calls made during the calculation.

ALGORITHM GCD(a, b, call_count)
INPUT:

- a, b: Two integers
- call_count: A list with a single integer to track number of recursive calls

OUTPUT:

- Greatest Common Divisor (GCD) of a and b
- Total number of recursive calls made

```
// Step 1: Define the recursive function
FUNCTION gcd(a, b, call_count)
```

```
    // Step 2: Increment recursive call count
    call_count[0] ← call_count[0] + 1

    // Step 3: Base Case
    IF b == 0 THEN
        RETURN a
    END IF

    // Step 4: Recursive Case
    RETURN gcd(b, a MOD b, call_count)
END FUNCTION

// Step 5: Initialize counter and call the function
LET a ← given integer (e.g., 48)
LET b ← given integer (e.g., 18)
LET call_count ← [0]          // List used to pass by reference
LET result ← gcd(a, b, call_count)

// Step 6: Output result
PRINT "The GCD of", a, "and", b, "is:", result
PRINT "Total recursive calls made:", call_count[0]

END ALGORITHM
```

Sample Input
a = 48, b = 18

Expected Output
The GCD of 48 and 18 is: 6
Total recursive calls made: 3

Time Complexity Analysis: O(log(min(a, b)))

Lab: 03

1. एक प्रोग्राम लिखें जो बाइनरी सर्च और टर्नरी सर्च तुलना किए गए कंपैरिज़न्स के आधार पर करे। इसे एक्सपेरिमेंटली ग्राफ प्लॉट करके और थ्योरिटिकली एल्गोरिदम रेंज सर्च का बाइनरी सर्च के साथ विश्लेषण करके करें।

Write a program to compare Binary search and Ternary search in terms of the number of comparisons done by each. Do this experimentally by plotting a graph and theoretically by analyzing the Algorithm range search using binary search

ALGORITHM Compare_Search_Methods(arr, target)
INPUT:

- A sorted array arr[0..n-1] of n elements
- A target value target to search for

OUTPUT:

- Index of target in array (if found)
- Number of comparisons for Binary and Ternary Search

Step 1: Binary Search with Comparison Count

```
LET low ← 0
LET high ← n - 1
LET comparisons_binary ← 0

WHILE low ≤ high DO
   comparisons_binary ← comparisons_binary + 1
   LET mid ← (low + high) // 2

   IF arr[mid] = target THEN
      RETURN mid
   ELSE IF arr[mid] < target THEN
      low ← mid + 1
   ELSE
      high ← mid - 1
   END IF
END WHILE

RETURN -1  // Target not found
```

Step 2: Ternary Search with Comparison Count

```
LET low ← 0
LET high ← n - 1
LET comparisons_ternary ← 0

WHILE low ≤ high DO
   LET mid1 ← low + (high - low) // 3
   LET mid2 ← high - (high - low) // 3

   comparisons_ternary ← comparisons_ternary + 1
   IF arr[mid1] = target THEN
```

```
      RETURN mid1
   comparisons_ternary ← comparisons_ternary + 1
   IF arr[mid2] = target THEN
      RETURN mid2

   IF target < arr[mid1] THEN
      high ← mid1 - 1
   ELSE IF target > arr[mid2] THEN
      low ← mid2 + 1
   ELSE
      low ← mid1 + 1
      high ← mid2 - 1
   END IF
END WHILE

RETURN -1  // Target not found
```

Step 3: Compare Number of Comparisons

```
Generate a sorted array arr of size n
Set target value

CALL Binary Search and record number of comparisons
CALL Ternary Search and record number of comparisons
PLOT a graph: input size (n) vs. number of comparisons
```

Step 4: Range Search Using Modified Binary Search

```
FUNCTION find_first(arr, target)
   LET low ← 0
   LET high ← n - 1
   LET result ← -1

   WHILE low ≤ high DO
      LET mid ← (low + high) // 2
      IF arr[mid] = target THEN
         result ← mid
         high ← mid - 1
      ELSE IF arr[mid] < target THEN
         low ← mid + 1
      ELSE
         high ← mid - 1
```

```
    END WHILE
    RETURN result

FUNCTION find_last(arr, target)
    LET low ← 0
    LET high ← n - 1
    LET result ← -1

    WHILE low ≤ high DO
        LET mid ← (low + high) // 2
        IF arr[mid] = target THEN
            result ← mid
            low ← mid + 1
        ELSE IF arr[mid] < target THEN
            low ← mid + 1
        ELSE
            high ← mid - 1
    END WHILE
    RETURN result

LET range ← find_last(arr, target) - find_first(arr, target) + 1
```

Sample Input
Sorted Array: [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
Target: 23

Expected Output
Binary Search:
Target found at index 5
Comparisons made: 3

Ternary Search:
Target found at index 5
Comparisons made: 4

Range Search:
Occurrences of target: 1

Time Complexity Analysis

Operation	Time Complexity

Binary Search	$O(\log_2 n)$
Ternary Search	$O(\log_3 n)$
Range Search (2 Binary)	$O(2 \cdot \log n) \approx O(\log n)$

2. एक सॉर्टेड ऐरे दिया गया है जिसमें अलग-अलग नॉन-नेगेटिव होल नंबर्स हैं। सबसे छोटा मिसिंग नंबर खोजें।

Given a sorted array of distinct non-negative whole numbers. Find the smallest missing number.

Hint: Binary search-based. Whole numbers are considered. If the number at the same index is greater than the index value, then something is missing in the left and right portions. Narrow down your search and reach the solution.

ALGORITHM Find_Smallest_Missing(arr, n)
INPUT:

- A sorted array arr[0..n-1] of n distinct non-negative integers
- Size of the array: n

OUTPUT:

- The smallest missing number from the array

Step 1: Initialize Pointers

```
plaintext
CopyEdit
LET low ← 0
LET high ← n - 1
```

Step 2: Binary Search Loop

```
plaintext
CopyEdit
WHILE low ≤ high DO
  LET mid ← (low + high) // 2

  IF arr[mid] = mid THEN
    // Missing number is in the right half
    low ← mid + 1
```

```
    ELSE
      // Missing number is in the left half or at mid
      high ← mid - 1
END WHILE
```

Step 3: Return the Missing Number

RETURN low // The smallest missing number

Explanation

- If arr[mid] == mid, it means all numbers till mid are present, so the missing number must be after mid.

- If arr[mid] > mid, the first missing number is somewhere before or at mid.

Sample Input
arr = [0, 1, 2, 3, 5]
n = 5

Expected Output
Smallest missing number is: 4

Time and Space Complexity: O(log n)

---

3. एक प्रोग्राम लिखें जो दिए गए n इंटीजर एलिमेंट्स को मर्ज सॉर्ट मेथड का उपयोग करके सॉर्ट करे और इसकी टाइम कॉम्प्लेक्सिटी की गणना करे। n > 5000 के विभिन्न मानों के लिए प्रोग्राम रन करें और सॉर्टिंग में लिया गया समय रिकॉर्ड करें। लिए गए समय और n के साइज के बीच एक ग्राफ प्लॉट करें। एलिमेंट्स को फाइल से पढ़ा जा सकता है या रैंडम नंबर जनरेटर का उपयोग करके उत्पन्न किया जा सकता है। किसी भी प्रोग्रामिंग लैंग्वेज का उपयोग करके यह प्रदर्शित करें कि डिवाइड एंड कॉन्कर मेथड कैसे काम करता है, साथ ही इसकी टाइम कॉम्प्लेक्सिटी एनालिसिस को समझाएं, जिसमें वर्स्ट केस, एवरेज केस,और बेस्ट केस शामिल हों।

Write a program to sort a given set of n integer elements using the Merge Sort method and compute its time complexity. Run the program for varied values of n > 5000, and record the time taken to sort. Plot a graph of the time taken versus the size of n. The elements can be read from a file or generated using a random number generator. Demonstrate using any language how the divide-and-conquer method works along with its time complexity analysis (worst case, average case, and best case).

ALGORITHM Merge_Sort(arr)
INPUT:

- An unsorted array arr[0..n-1] of length n

OUTPUT:

- A sorted version of the input array

Step 1: Base Case

IF length of arr ≤ 1 THEN
  RETURN arr // Already sorted

Step 2: Divide the Array

LET mid ← length(arr) // 2
LET left ← Merge_Sort(arr[0..mid-1])
LET right ← Merge_Sort(arr[mid..n-1])

Step 3: Merge Sorted Halves
Call Merge(left, right) to combine the two sorted halves:

```
FUNCTION Merge(left, right):
  LET result ← empty list
  LET i ← 0, j ← 0

  WHILE i < length(left) AND j < length(right) DO
    IF left[i] < right[j] THEN
      APPEND left[i] to result
      i ← i + 1
    ELSE
      APPEND right[j] to result
      j ← j + 1
  END WHILE

  APPEND remaining elements of left[i..] to result
  APPEND remaining elements of right[j..] to result

  RETURN result
```

Step 4: Return Final Merged List

RETURN Merge(left, right)

Sample Input
arr = [12, 11, 13, 5, 6, 7]

Expected Output
Sorted array is: [5, 6, 7, 11, 12, 13]

Time Complexity: O(nlog n)

Lab: 04

1. एक प्रोग्राम लिखें जो दिए गए नंबर्स को सेलेक्शन सॉर्ट एल्गोरिदम का उपयोग करके सॉर्ट करे। अलग-अलग n -एलिमेंट्स की संख्या के लिए एक्सपेरिमेंट दोहराएं और लिए गए समय बनाम n के बीच एक ग्राफ प्लॉट करें। एलिमेंट्स को फाइल से पढ़ा जा सकता है या रैंडम नंबर जनरेटर का उपयोग करके उत्पन्न किया जा सकता है।

Write a program to sort a given set of numbers using the Selection Sort algorithm. Repeat the experiment for different values of n (number of elements in the list to be sorted) and plot a graph of the time taken versus n. The elements can be read from a file or generated using a random number generator.

ALGORITHM Merge_Sort_LinkedList(head)
INPUT:
A singly linked list with head node head
OUTPUT:
A sorted linked list in ascending order

Step 1: Base Case

IF head is NULL OR head.next is NULL THEN
  RETURN head // Already sorted

Step 2: Divide the Linked List

LET middle ← Get_Middle(head)
LET next_to_middle ← middle.next
SET middle.next ← NULL // Split the list into two halves

LET left ← Merge_Sort_LinkedList(head)
LET right ← Merge_Sort_LinkedList(next_to_middle)

Step 3: Merge the Sorted Halves

CALL Merge_LinkedLists(left, right)

FUNCTION Merge_LinkedLists(left, right):

```
sql
CopyEdit
IF left is NULL THEN
   RETURN right
IF right is NULL THEN
   RETURN left

IF left.value < right.value THEN
   left.next ← Merge_LinkedLists(left.next, right)
   RETURN left
ELSE
   right.next ← Merge_LinkedLists(left, right.next)
   RETURN right
```

Step 4: Return Final Merged List

RETURN Merge_LinkedLists(left, right)

Helper Function: Get_Middle(head)

LET slow ← head, fast ← head

WHILE fast.next ≠ NULL AND fast.next.next ≠ NULL DO
   slow ← slow.next
   fast ← fast.next.next

RETURN slow

Sample Input

A linked list:

head = 12 → 11 → 13 → 5 → 6 → 7

Expected Output

Sorted linked list:

5 → 6 → 7 → 11 → 12 → 13

Time Complexity

Best Case: O(n log n)
Average Case: O(n log n)
Worst Case: O(n log n)

---

2. एक प्रोग्राम लिखें जो डिफेक्टिव कॉइन प्रॉब्लम को हल करे, जिसमें दिए गए सिक्कों के समूह में यदि कोई डिफेक्टिव कॉइन मौजूद हो, तो उसे पहचानना और उसके डिफेक्ट (हल्का/भारी) की प्रकृति निर्धारित करना शामिल है। यह प्रक्रिया डिवाइड एंड कॉन्कर अप्रोच का उपयोग करके की जानी चाहिए, जहां अधिकतम एक डिफेक्टिव कॉइन मौजूद हो सकता है।

Write a program to solve the defective coin problem, which involves identifying any defective coin, if present, and determining the nature of the defect (lighter/heavier) from a set of coins containing at most one defective coin using the divide and conquer approach.

ALGORITHM Find_Defective_Coin(coins)
INPUT: A list coins[0..n-1] of coin weights, where all coins are identical except one (either heavier or lighter)
OUTPUT:
The weight of the defective coin

Step 1: Define Recursive Helper Function

```
FUNCTION Find(coin_set):
   IF length(coin_set) == 1 THEN
      RETURN coin_set[0]  // Base case: only one coin, must be defective
```

Step 2: Divide the Set into Three Parts

```
LET third ← floor(length(coin_set) / 3)
LET part1 ← coin_set[0..third-1]
LET part2 ← coin_set[third..2*third-1]
LET part3 ← coin_set[2*third..end]
```

Step 3: Compare the Weights

```
IF sum(part1) < sum(part2) THEN
   RETURN Find(part1)  // Defective coin is lighter and in part1
ELSE IF sum(part1) > sum(part2) THEN
   RETURN Find(part2)  // Defective coin is heavier and in part2
ELSE
   RETURN Find(part3)  // Both part1 and part2 are equal, defect is in part3
```

Step 4: Call the Function

RETURN Find(coins)

Sample Input
coins = [1, 1, 1, 1, 1, 1, 2]

Expected Output
Defective Coin: 2

Time Complexity
Best Case: $O(\log_3 n)$
Average Case: $O(\log_3 n)$
Worst Case: $O(\log_3 n)$

Lab: 05

1. दिए गए n पूर्णांक तत्वों के सेट को क्विक सॉर्ट मेथड का उपयोग करके सॉर्ट करें और इसकी टाइम कॉम्प्लेक्सिटी की गणना करें। प्रोग्राम को n>5000 के विभिन्न मानों के लिए चलाएं और सॉर्ट करने में लिया गया समय रिकॉर्ड करें। लिए गए समय बनाम n का ग्राफ गैर ग्राफ शीट पर प्लॉट करें। तत्वों को फाइल से पढ़ा जा सकता है या रैंडम नंबर जनरेटर का उपयोग करके उत्पन्न किया जा सकता है। किसी भी भाषा का उपयोग करके प्रदर्शित करें कि डिवाइड एंड कॉन्कर मेथड कैसे काम करता है, साथ ही इसकी टाइम कॉम्प्लेक्सिटी का विश्लेषण करें: वर्स्ट केस, एवरेज केस और बेस्ट केस।

Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus a non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate using any language how the divide and-conquer method works along with its time complexity analysis: worst case, average case and best case.

a) एक प्रोग्राम लिखें जो क्विक सॉर्ट को रैंडम पिवट का उपयोग करके इंप्लीमेंट करे। इसकी टाइम कॉम्प्लेक्सिटी का विश्लेषण करें जब इनपुट रिवर्स सॉर्टेड, सॉर्टेड और रैंडम ऐरे हो।

Write a program to implement Quick Sort using random pivot. Analyse its time complexity for reverse sorted, sorted and random array.

ALGORITHM Quick_Sort_Median_of_Three(arr, low, high)
INPUT:
An unsorted array arr[low..high] of length n
OUTPUT:
A sorted version of the input array

Step 1: Base Case
pgsql
CopyEdit
IF low < high THEN
  CONTINUE
ELSE
  RETURN // Sub-array of size 0 or 1 is already sorted

Step 2: Choose Pivot using Median of Three
LET mid ← (low + high) // 2
LET a ← arr[low], b ← arr[mid], c ← arr[high]
Sort a, b, c and select the **median** as pivot
Place the pivot at arr[high] (swap with original high if needed)

Step 3: Partition the Array
LET pivot ← arr[high]
LET i ← low - 1

FOR j FROM low TO high - 1 DO
  IF arr[j] < pivot THEN
    i ← i + 1
    SWAP arr[i] and arr[j]
SWAP arr[i+1] and arr[high]
LET partition_index ← i + 1

Step 4: Recur on Sub-arrays
CALL Quick_Sort_Median_of_Three(arr, low, partition_index - 1)
CALL Quick_Sort_Median_of_Three(arr, partition_index + 1, high)

Sample Input
arr = [24, 3, 45, 29, 37, 1, 18, 50]

Expected Output
Sorted array: [1, 3, 18, 24, 29, 37, 45, 50]

Time Complexity
Best Case: O(n log n)

Average Case: O(n log n)
Worst Case: O(n^2) (less likely due to pivot choice optimization)

---

b) एक प्रोग्राम लिखें जो क्विक सॉर्ट को इंसरशन सॉर्ट का उपयोग करके इंप्लीमेंट करे (जब n > 4 हो तो क्विक सॉर्ट और जब n ≤ 4 हो तो इंसरशन सॉर्ट)।

Write a program to implement Quick Sort using insertion sort (for n > 4, quick sort N<= 4, insertion sort)

ALGORITHM Quick_Sort_With_Insertion(arr, low, high)

INPUT:
An unsorted array arr[low..high] of length n
OUTPUT:
A sorted version of the input array

Step 1: Base Case
IF (high - low + 1) ≤ 4 THEN
  CALL Insertion_Sort(arr, low, high)
  RETURN

Step 2: Quick Sort Partitioning
LET pivot ← arr[high]
LET i ← low - 1

FOR j FROM low TO high - 1 DO
  IF arr[j] < pivot THEN
    i ← i + 1
    SWAP arr[i], arr[j]
SWAP arr[i+1], arr[high]
LET partition_index ← i + 1

Step 3: Recur on Sub-arrays
CALL Quick_Sort_With_Insertion(arr, low, partition_index - 1)
CALL Quick_Sort_With_Insertion(arr, partition_index + 1, high)

Helper Algorithm: Insertion Sort
ALGORITHM Insertion_Sort(arr, low, high)
FOR i FROM low+1 TO high DO
  LET key ← arr[i]

```
LET j ← i - 1
WHILE j ≥ low AND arr[j] > key DO
   arr[j+1] ← arr[j]
   j ← j - 1
arr[j+1] ← key
```

Sample Input
arr = [29, 3, 7, 18, 1, 45, 24, 4]

Expected Output
Sorted array: [1, 3, 4, 7, 18, 24, 29, 45]

Time Complexity
Quick Sort (n > 4):
- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n^2)

Insertion Sort (n ≤ 4):
- Best, Average, Worst Case: O(n^2)

Lab: 06

1. डायनेमिक प्रोग्रामिंग का उपयोग करके 0/1 नैपसैक समस्या को लागू करने के लिए एल्गोरिदम लिखें।
Write the algorithm to implement 0/1 knapsack problem using dynamic programming.

ALGORITHM Knapsack_DP(n, W, w, v)
INPUT: Number of items n, knapsack capacity W, array of weights w[1..n], array of values v[1..n]
OUTPUT: Maximum value achievable within capacity W

```
// Step 1: Initialize DP table
LET DP[0..n][0..W] ← 0
// Step 2: Populate DP table
FOR i ← 1 TO n DO
   FOR j ← 0 TO W DO
      IF w[i] ≤ j THEN
         DP[i][j] ← MAX(DP[i-1][j], DP[i-1][j - w[i]] + v[i])
      ELSE
         DP[i][j] ← DP[i-1][j]
      END IF
   END FOR
```

```
    END FOR
    // Step 3: Return the result
    RETURN DP[n][W]

END ALGORITHM
```

Sample Input

- **Number of items**: n = 4
- **Capacity of the knapsack**: W = 8
- **Weights of the items**: w = [2, 3, 4, 5]
- **Values of the items**: v = [3, 4, 5, 6]

Expected Output

- **Maximum value that can be obtained**: 9

Time Complexity: O(nW) Where n is the number of items and W is the maximum weight the knapsack can carry.

---

2. स्ट्रैसेन के मैट्रिक्स गुणन को लागू करने के लिए एक प्रोग्राम लिखें।
Write an algorithm to implement Strassen's matrix multiplication for both a constant value of n and a dynamic value of n.

Strassen's Algorithm for Constant N:

```
ALGORITHM Strassen_Multiply(A, B)
    INPUT: Two square matrices A and B of size n × n (where n is a power of 2)
    OUTPUT: Matrix C, the product of A and B
    IF n = 1 THEN
        RETURN [[A[1,1] * B[1,1]]]
    END IF
    LET mid ← n / 2
    // Divide matrices into submatrices
    (A11, A12, A21, A22) ← Split(A)
    (B11, B12, B21, B22) ← Split(B)
    // Compute 7 matrix products using Strassen's formula
    M1 ← Strassen_Multiply(Add(A11, A22), Add(B11, B22))
    M2 ← Strassen_Multiply(Add(A21, A22), B11)
    M3 ← Strassen_Multiply(A11, Subtract(B12, B22))
    M4 ← Strassen_Multiply(A22, Subtract(B21, B11))
```

```
    M5 ← Strassen_Multiply(Add(A11, A12), B22)
    M6 ← Strassen_Multiply(Subtract(A21, A11), Add(B11, B12))
    M7 ← Strassen_Multiply(Subtract(A12, A22), Add(B21, B22))
    // Combine results to form C
    C11 ← Add(Subtract(Add(M1, M4), M5), M7)
    C12 ← Add(M3, M5)
    C21 ← Add(M2, M4)
    C22 ← Add(Subtract(Add(M1, M3), M2), M6)
    C ← Combine(C11, C12, C21, C22)
    RETURN C
END ALGORITHM
```

Strassen's Algorithm for Dynamic N:

```
ALGORITHM Strassen_Dynamic(A, B)
    INPUT: Two matrices A and B of size n × m and m × p
    OUTPUT: Matrix C, the product of A and B
    LET max_dim ← MAX(n, m, p)
    LET k ← Next_Power_Of_Two(max_dim)
    // Pad matrices A and B to size k × k
    A_padded ← Pad_Matrix(A, k)
    B_padded ← Pad_Matrix(B, k)
    // Perform Strassen's multiplication on padded matrices
    C_padded ← Strassen_Multiply(A_padded, B_padded)
    // Extract the result of original size from the padded matrix
    C ← Extract_Submatrix(C_padded, n, p)
    RETURN C
END ALGORITHM

ALGORITHM Next_Power_Of_Two(x)
    INPUT: An integer x
    OUTPUT: The smallest power of 2 greater than or equal to x
    LET power ← 1
    WHILE power < x DO
        power ← power * 2
    END WHILE
    RETURN power
END ALGORITHM

ALGORITHM Pad_Matrix(M, size)
```

```
    INPUT: A matrix M and the desired size
    OUTPUT: A padded matrix of size × size
    LET n, m ← Dimensions(M)
    LET P ← A zero matrix of size × size
    FOR i ← 1 TO n DO
      FOR j ← 1 TO m DO
        P[i, j] ← M[i, j]
      END FOR
    END FOR
    RETURN P
END ALGORITHM
```

ALGORITHM Extract_Submatrix(M, rows, cols)

```
    INPUT: A matrix M and dimensions rows × cols
    OUTPUT: The submatrix of M of size rows × cols
    LET S ← A zero matrix of rows × cols
    FOR i ← 1 TO rows DO
      FOR j ← 1 TO cols DO
        S[i, j] ← M[i, j]
      END FOR
    END FOR
    RETURN S
END ALGORITHM
```

Helper Algorithms:

Add: Matrix addition.
Subtract: Matrix subtraction.
Split: Splits a matrix into four submatrices.
Combine: Combines four submatrices into one.

ALGORITHM Add(A, B)

```
    INPUT: Two matrices A and B of the same size
    OUTPUT: Matrix C, where C[i, j] = A[i, j] + B[i, j]
    LET n ← Number of rows in A
    LET m ← Number of columns in A
    LET C ← A zero matrix of size n × m
    FOR i ← 1 TO n DO
      FOR j ← 1 TO m DO
        C[i, j] ← A[i, j] + B[i, j]
      END FOR
    END FOR
    RETURN C
END ALGORITHM
```

ALGORITHM Subtract(A, B)
INPUT: Two matrices A and B of the same size
OUTPUT: Matrix C, where C[i, j] = A[i, j] - B[i, j]
LET n ← Number of rows in A
LET m ← Number of columns in A
LET C ← A zero matrix of size n × m
FOR i ← 1 TO n DO
FOR j ← 1 TO m DO
C[i, j] ← A[i, j] - B[i, j]
END FOR
END FOR
RETURN C
END ALGORITHM

ALGORITHM Split(M)
INPUT: A matrix M of size n × n
OUTPUT: Submatrices M11, M12, M21, M22 of size n/2 × n/2
LET mid ← n / 2
M11 ← Submatrix of M[1:mid, 1:mid]
M12 ← Submatrix of M[1:mid, mid+1:n]
M21 ← Submatrix of M[mid+1:n, 1:mid]
M22 ← Submatrix of M[mid+1:n, mid+1:n]
RETURN (M11, M12, M21, M22)
END ALGORITHM

ALGORITHM Combine(M11, M12, M21, M22)
INPUT: Four matrices M11, M12, M21, M22 of the same size
OUTPUT: A combined matrix C
LET n ← Size of M11
LET C ← A zero matrix of size 2n × 2n
FOR i ← 1 TO n DO
FOR j ← 1 TO n DO
C[i, j] ← M11[i, j]
C[i, j+n] ← M12[i, j]
C[i+n, j] ← M21[i, j]
C[i+n, j+n] ← M22[i, j]
END FOR
END FOR
RETURN C
END ALGORITHM

Sample Input

- **Matrix A**:
- **Matrix B**:

Expected Output

- **Matrix C (Product of A and B)**:

Time Complexity: O(n ^ (2.81))

---

3. एक एल्गोरिदम लिखें जो डिवाइड एंड कॉन्कर का उपयोग करके लॉन्ग इंटीजर्स के मल्टिप्लिकेशन को इंप्लीमेंट करे।

Write an algorithm to implement the multiplication of long integers using divide and conquer.

ALGORITHM Long_Integer_Multiply(X, Y)
INPUT: Two long integers X and Y, both of size n
OUTPUT: The product Z = X × Y
// Base case: if the integers are small enough, multiply them directly
IF n = 1 THEN
RETURN X × Y
END IF
// Divide the integers into halves
LET m ← ⌊n / 2⌋ // Middle point
LET X1 ← Higher_Half(X, m)
LET X0 ← Lower_Half(X, m)
LET Y1 ← Higher_Half(Y, m)
LET Y0 ← Lower_Half(Y, m)
// Recursively compute the three products
Z2 ← Long_Integer_Multiply(X1, Y1)
Z0 ← Long_Integer_Multiply(X0, Y0)
Z1 ← Long_Integer_Multiply(X1 + X0, Y1 + Y0) - Z2 - Z0
// Combine results
Z ← (Z2 × 10^(2m)) + (Z1 × 10^m) + Z0
RETURN Z
END ALGORITHM

ALGORITHM Higher_Half(X, m)
INPUT: Integer X and size m

OUTPUT: The higher half of the integer X (leftmost digits)
RETURN Floor(X / 10^m)
END ALGORITHM

ALGORITHM Lower_Half(X, m)
INPUT: Integer X and size m
OUTPUT: The lower half of the integer X (rightmost digits)
RETURN X MOD 10^m
END ALGORITHM

Sample Input

- **Integer X**: 1234
- **Integer Y**: 5678

Expected Output

- **Product Z = X × Y**: 7006652

Time Complexity: O(nlog23)≈O(n1.58)

Lab: 07

1. एक दिए गए वर्टेक्स से एक वेटेड कनेक्टेड ग्राफ में अन्य वर्टिस तक की शॉर्टेस्ट पाथ्स को खोजें, डाइकस्ट्रा एल्गोरिदम का उपयोग करके। किसी भी भाषा का उपयोग करके प्रोग्राम लिखें।

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Write the program using any language.

ALGORITHM Dijkstra's_Algorithm(graph, startVertex)
INPUT: A directed weighted graph and a start vertex
OUTPUT: The shortest distances from startVertex to all other vertices
// Initialization
LET distanceArray ← Initialize_Distance_Array(graph, startVertex)
LET distanceRecord ← Create_Priority_Queue()
LET visitedVertices ← Create_Empty_List()
// Add start vertex to priority queue
ENQUEUE distanceRecord, distanceArray[startVertex.Index]
// Main Loop
WHILE visitedVertices.Count ≠ distanceArray.Length AND distanceRecord.Count ≠ 0

```
// Dequeue vertex with minimum distance
LET minDistance ← DEQUEUE distanceRecord

// Skip if vertex is already visited
IF visitedVertices.Contains(minDistance.Vertex)
   CONTINUE
END IF
// Mark vertex as visited
ADD visitedVertices, minDistance.Vertex
// Update distances of neighbors
FOR EACH neighbor IN graph.GetNeighbors(minDistance.Vertex)
   LET adjacentDistance ← graph.AdjacentDistance(minDistance.Vertex, neighbor)
   LET distance ← distanceArray[neighbor.Index]
   LET fullDistance ← minDistance.Distance + adjacentDistance
   IF distance.Distance > fullDistance
      UPDATE distance.Distance, fullDistance
      UPDATE distance.PreviousVertex, minDistance.Vertex
      ENQUEUE distanceRecord, distance
   END IF
END FOR
END WHILE
// Return shortest distances
RETURN distanceArray
END ALGORITHM
```

ALGORITHM Initialize_Distance_Array(graph, startVertex)

```
INPUT: A directed weighted graph graph and a start vertex startVertex
OUTPUT: The initialized distance array
LET distanceArray ← Create_Array(graph.Count)
// Initialize start vertex distance to 0
SET distanceArray[startVertex.Index].Distance, 0
// Initialize all other vertex distances to infinity
FOR EACH vertex IN graph.Vertices
IF vertex ≠ startVertex
SET distanceArray[vertex.Index].Distance, double.MaxValue
END IF
END FOR
RETURN distanceArray
END ALGORITHM
```

Sample Input:

```
vector<vector<pair<int, int>>> graph = {
   {{1, 10}, {3, 5}}, // Edges from vertex 0
```

```
    {{2, 1}},       // Edges from vertex 1
    {{3, 1}},       // Edges from vertex 2
    {{2, 1}}        // Edges from vertex 3
};
```

Expected Output
Vertex 0 -> Distance 0
Vertex 1 -> Distance 10
Vertex 2 -> Distance 6
Vertex 3 -> Distance 5

Time Complexity: Using a min-heap, the time complexity is O((V+E)log⁡V)O((V + E) \log V) where VV is the number of vertices and EE is the number of edges.

---

2. एक प्रोग्राम लिखें जो हीप्स का उपयोग करके ग्राफ (E, V) में एमएसटी के लिए प्रिम्स मेथड को इंप्लीमेंट करे।

Write a program to implement Prim's method for MST in a graph (E, V) using heaps.

ALGORITHM Prim's_MST(graph)
INPUT: A weighted graph graph with V vertices
OUTPUT: The minimum spanning tree (MST) of the graph
// Define a struct to store MST information
STRUCT mst {
bool visited;
int key;
int near;
}
// Initialize MST array
LET MST_Array ← Create_Array(V)
// Initialize MST array values
PROCEDURE Initialize()
FOR i FROM 0 TO V-1
MST_Array[i].visited ← false
MST_Array[i].key ← INFINITY
MST_Array[i].near ← i
MST_Array[0].key ← 0
// Update near vertices
PROCEDURE Update_Near()
FOR v FROM 0 TO V-1
LET min ← INFINITY

```
LET minIndex ← 0
FOR i FROM 0 TO V-1
IF MST_Array[i].key < min AND MST_Array[i].visited = false AND MST_Array[i].key
≠ INFINITY
min ← MST_Array[i].key
minIndex ← i
MST_Array[minIndex].visited ← true
FOR i FROM 0 TO V-1
IF graph[minIndex][i] ≠ 0 AND graph[minIndex][i] < INFINITY
IF graph[minIndex][i] < MST_Array[i].key
MST_Array[i].key ← graph[minIndex][i]
MST_Array[i].near ← minIndex
// Show MST
PROCEDURE Show()
FOR i FROM 0 TO V-1
PRINT i, " - ", MST_Array[i].near, "\t", graph[i][MST_Array[i].near]
// Main procedure
PROCEDURE Main()
Initialize()
Update_Near()
Show()
END ALGORITHM
```

Sample Input:

```
const int V = 6;
int graph[V][V] = {
   {0, 2, 0, 6, 0, 0},
   {2, 0, 3, 8, 0, 5},
   {0, 3, 0, 7, 0, 0},
   {6, 8, 7, 0, 9, 0},
   {0, 0, 0, 9, 0, 10},
   {0, 5, 0, 0, 10, 0}
};
```

Expected Output:

0 - 1 2
1 - 2 3
0 - 3 6
1 - 5 5
3 - 4 9

Time Complexity: Using a min-heap, the time complexity is O(Elog⁡V)O(E \log V).

---

3. एक प्रोग्राम लिखें जो हीप्स का उपयोग करके क्रूस्कल्स एल्गोरिदम द्वारा ग्राफ का मिनिमम कॉस्ट स्पैनिंग ट्री इंप्लीमेंट करे।

Write a program to implement a minimum cost spanning tree of a graph using Kruskal's algorithm using heaps.

```
ALGORITHM Kruskal's_MST(graph)
INPUT: A weighted graph graph
OUTPUT: The minimum spanning tree (MST) of the graph

// Step 1: Create a disjoint set
PROCEDURE Create_Disjoint_Set()
LET disjoint_set ← Create_Disjoint_Set_Tree()
FOR EACH node IN graph.connections
disjoint_set.make_set(node)

// Step 2: Get the edges in ascending order of weights
PROCEDURE Get_Edges()
LET edges ← Create_Empty_List()
LET seen ← Create_Empty_Set()
FOR EACH start IN graph.connections
FOR EACH end IN graph.connections[start]
IF (start, end) NOT IN seen
seen.add((end, start))
edges.append((start, end, graph.connections[start][end]))
edges.sort(key=lambda x: x[2])

// Step 3: Generate the MST
PROCEDURE Generate_MST()
LET num_edges ← 0
LET index ← 0
LET graph_mst ← Create_Empty_Graph()
WHILE num_edges < len(graph.connections) - 1
u, v, w ← edges[index]
index ← index + 1
parent_u ← disjoint_set.find_set(u)
parent_v ← disjoint_set.find_set(v)
IF parent_u ≠ parent_v
```

num_edges ← num_edges + 1
graph_mst.add_edge(u, v, w)
disjoint_set.union(u, v)
RETURN graph_mst

END ALGORITHM

Sample Input:

```
const int V = 6;
vector<tuple<int, int, int>> edges = {
   {0, 1, 2}, {0, 3, 6}, {0, 4, 8}, {1, 2, 3},
   {1, 3, 8}, {1, 5, 5}, {2, 3, 7}, {3, 4, 9},
   {4, 5, 10}
};
```

Expected Output:

Edge 0 - 1, Weight: 2
Edge 1 - 2, Weight: 3
Edge 0 - 3, Weight: 6
Edge 1 - 5, Weight: 5
Edge 3 - 4, Weight: 9

Time Complexity: Using union-find (disjoint-set) and heaps, the time complexity is O(Elog⁡E)O(E \log E) or O(Elog⁡V)O(E \log V).

LAB 8

1. एक प्रोग्राम लिखें जो ग्रीडी एल्गोरिदम का उपयोग करके चेंज-मेकिंग प्रॉब्लम को इंप्लीमेंट करे, जब (1)कैशियर के पास केवल सीमित संख्या में कॉइन्स हों। (2)कैशियर के पास अनलिमिटेड सप्लाई में कॉइन्स हों। कॉइन्स के मूल्य (1, 5, 10, 20, 50... पैसे) जैसे होंगे।

Write a program to implement Change – making problem using a greedy algorithm when (1) the cashier has only a limited number of coins. (2) the cashier has an unlimited supply of coins. The value of coins will be like (1,5,10,20,50…paise)

ALGORITHM Greedy_Coin_Change(amount, denominations)
INPUT: Amount to change and list of coin denominations
OUTPUT: Change in coins or message if change cannot be made exactly
// Sort denominations in descending order
denominations.sort(reverse=True)

```
// Initialize change dictionary
LET change ← Create_Empty_Dictionary()
// Iterate through each denomination
FOR EACH coin IN denominations
IF amount ≥ coin
// Calculate maximum coins of this denomination
LET num_coins ← amount // coin
change[coin] ← num_coins
amount ← amount - num_coins * coin
// Check if change can be made exactly
IF amount > 0
RETURN "Change cannot be made exactly."
ELSE
RETURN change
```

ALGORITHM Greedy_Coin_Change_Limited(amount, denominations, available_coins)

INPUT: Amount to change, list of coin denominations, and dictionary of available coins
OUTPUT: Change in coins or message if change cannot be made exactly with available coins

```
// Sort denominations in descending order
denominations.sort(reverse=True)
// Initialize change dictionary
LET change ← Create_Empty_Dictionary()
// Iterate through each denomination
FOR EACH coin IN denominations
IF amount ≥ coin AND available_coins[coin] > 0
// Calculate maximum coins of this denomination
LET num_coins ← min(amount // coin, available_coins[coin])
change[coin] ← num_coins
amount ← amount - num_coins * coin
available_coins[coin] ← available_coins[coin] - num_coins
// Check if change can be made exactly with available coins
IF amount > 0
RETURN "Change cannot be made exactly with available coins."
ELSE
RETURN change
```

1. **Greedy Coin Change (Unlimited Coins)**

Sample Input

- **Amount**: 93
- **Denominations**: [1, 5, 10, 20, 50]

Expected Output

- **Change**: {50: 1, 20: 2, 1: 3}

2. Greedy Coin Change (Limited Coins)

Sample Input

- **Amount**: 93
- **Denominations**: `[1, 5, 10, 20, 50]`
- **Available Coins**:

{1, 5}, // 5 coins of 1 paise
{5, 2}, // 2 coins of 5 paise
{10, 2}, // 2 coins of 10 paise
{20, 1}, // 1 coin of 20 paise
{50, 1} // 1 coin of 50 paise

Expected Output

- Change: Change cannot be made exactly with available coins.

Limited Number of Coins Time Complexity: Assuming n is the number of distinct coin denominations and m is the total amount, the time complexity would be $O(n \cdot m)$.

Unlimited Supply of Coins Time Complexity: The time complexity is O(n) where n is the number of different coin denominations.

---

2. दिज्क्स्ट्रा के एल्गोरिदम का उपयोग करके डायरेक्टेड एसाइक्लिक ग्राफ़ (डीएजी) में सिंगल सोर्स शॉर्टेस्ट पाथ (एसएसएसपी) खोजने के लिए एक प्रोग्राम लिखें।

Write a program to find the Single Source Shortest Path (SSSP) in a Directed Acyclic Graph (DAG) using Dijkstra's Algorithm. heap, linear list

ALGORITHM Dijkstra_Heap(graph, start)
INPUT: Weighted graph graph and start node start
OUTPUT: Shortest distances from start node to all other nodes
// Initialize heap with start node
LET heap ← Create_Heap()
heap.push((0, start))
// Initialize distances dictionary
LET distances ← Create_Dictionary()
FOR EACH node IN graph
distances[node] ← ∞
distances[start] ← 0

```
// Dijkstra's algorithm
WHILE heap IS NOT EMPTY
current_distance, current_node ← heap.pop()
IF current_distance > distances[current_node]
CONTINUE
FOR EACH neighbor, weight IN graph[current_node]
distance ← current_distance + weight
IF distance < distances[neighbor]
distances[neighbor] ← distance
heap.push((distance, neighbor))
RETURN distances
```

ALGORITHM Dijkstra_Linear(graph, start)

```
INPUT: Weighted graph graph and start node start
OUTPUT: Shortest distances from start node to all other nodes
// Initialize unvisited list
LET unvisited ← Create_List()
FOR EACH node IN graph
unvisited.add(node)
// Initialize distances dictionary
LET distances ← Create_Dictionary()
FOR EACH node IN graph
distances[node] ← ∞
distances[start] ← 0
// Dijkstra's algorithm
WHILE unvisited IS NOT EMPTY
current_node ← min(unvisited, key=lambda node: distances[node])
unvisited.remove(current_node)
FOR EACH neighbor, weight IN graph[current_node]
distance ← distances[current_node] + weight
IF distance < distances[neighbor]
distances[neighbor] ← distance
RETURN distances
```

Sample Input:

```
vector<vector<pair<int, int>>> graph = {
   {{1, 1}, {2, 4}, {3, 7}}, // Edges from vertex 0
   {{0, 1}, {2, 3}, {3, 5}}, // Edges from vertex 1
   {{0, 4}, {1, 3}, {3, 6}}, // Edges from vertex 2
   {{0, 7}, {1, 5}, {2, 6}}  // Edges from vertex 3
};
```

Expected Output

Vertex 0 -> Distance 0
Vertex 1 -> Distance 1
Vertex 2 -> Distance 4
Vertex 3 -> Distance 6

Time Complexity: Using a min-heap (priority queue), the time complexity is O((V+E)log$_{f0}$V)

LAB - 09

1. एक प्रोग्राम लिखें जो विभिन्न लंबाइयों की सॉर्टेड फाइल्स के सेट को कुशलतापूर्वक मर्ज करके एक सिंगल सॉर्टेड फाइल बनाए, ताकि प्रोसेस की टाइम कॉम्प्लेक्सिटी को न्यूनतम किया जा सके।

Write a program that efficiently merges a set of sorted files of different lengths into a single sorted file, aiming to minimize the time complexity of the process.

ALGORITHM Merge_Sorted_Files(files)
INPUT: A list of sorted files, where each file is a list of integers
OUTPUT: A single sorted list of integers, merged from all input files
// Step 1: Initialize a min-heap to store elements from each file
CREATE a min-heap, minHeap, to store FileNode elements
// Step 2: Initialize indices to track positions in each file
CREATE a list, indices, of size files.size(), initialized to 0
// Step 3: Push first element from each file into the min-heap
FOR EACH file IN files
  IF file IS NOT EMPTY
    CREATE a FileNode element, node, with value = file[0] and fileIndex = index of file
    PUSH node INTO minHeap
// Step 4: Extract the smallest element and insert the next from the same file
CREATE a list, result, to store the merged sorted file
WHILE minHeap IS NOT EMPTY
  EXTRACT the minimum element, node, FROM minHeap
  APPEND node.value TO result
  INCREMENT indices[node.fileIndex]
  IF indices[node.fileIndex] < length of file[node.fileIndex]
   CREATE a new FileNode element, newNode, with value = file[node.fileIndex][indices[node.fileIndex]] and fileIndex = node.fileIndex
    PUSH newNode INTO minHeap
RETURN result
END ALGORITHM

Sample Input:
files = [
[1, 4, 7],
[2, 5, 8],
[3, 6, 9]
]

Expected Output:
[1, 2, 3, 4, 5, 6, 7, 8, 9]

Time Complexity: If there are k sorted files with a total of n elements, the time complexity is O(nlogfok)O(n \log k).

—---

2. हफ़मैन कोडिंग को लागू करने के लिए एक प्रोग्राम लिखें। प्रोग्राम को हफमैन एल्गोरिदम का उपयोग करके दिए गए इनपुट टेक्स्ट को एनकोड करना चाहिए, प्रत्येक कैरेक्टर के लिए संबंधित कोड जेनरेट करना चाहिए। इसे हफमैन-एनकोडेड टेक्स्ट को उसके ओरिजिनल फॉर्म में डिकोड करने में भी सक्षम होना चाहिए, और वेरिएबल-लेंथ एनकोडिंग और डिकोडिंग को कुशलतापूर्वक हैंडल करना चाहिए।

WAP to implement Huffman coding. The program should encode a given input text using Huffman's algorithm, generating corresponding codes for each character. It should also decode Huffman-encoded text back to its original form, efficiently handling variable-length encoding and decoding.

ALGORITHM Huffman_Coding(text)
INPUT: Input text to be encoded
OUTPUT: Encoded and decoded text using Huffman coding

// Step 1: Calculate frequency of each character
CREATE a frequency map, freqMap, to store frequency of each character
FOR EACH character IN text
IF character IS NOT IN freqMap
ADD character TO freqMap WITH frequency 1
ELSE
INCREMENT frequency OF character IN freqMap

// Step 2: Build Huffman tree
CREATE a Huffman tree, root, using buildHuffmanTree function
FUNCTION buildHuffmanTree(freqMap)
CREATE a priority queue, minHeap, to store Huffman nodes
FOR EACH character-frequency pair IN freqMap

```
CREATE a new Huffman node, node, with character and frequency
PUSH node INTO minHeap
WHILE minHeap size > 1
EXTRACT the minimum frequency node, left, FROM minHeap
EXTRACT the minimum frequency node, right, FROM minHeap
CREATE a new internal node, newNode, with frequency = left frequency + right frequency
newNode left child = left
newNode right child = right
PUSH newNode INTO minHeap
RETURN minHeap top node

// Step 3: Generate Huffman codes
CREATE a Huffman code map, huffmanCode, to store Huffman codes for each character
FUNCTION generateHuffmanCodes(root, code, huffmanCode)
IF root IS NULL
RETURN
IF root character IS NOT NULL
ADD root character TO huffmanCode WITH code
RECURSIVELY CALL generateHuffmanCodes FOR root left child WITH code + "0"
RECURSIVELY CALL generateHuffmanCodes FOR root right child WITH code + "1"

// Step 4: Encode input text
CREATE an encoded string, encodedText, to store encoded input text
FUNCTION encode(text, huffmanCode)
FOR EACH character IN text
APPEND huffmanCode[character] TO encodedText
RETURN encodedText

// Step 5: Decode Huffman encoded text
CREATE a decoded string, decodedText, to store decoded encoded text
FUNCTION decode(root, encodedText)
FOR EACH bit IN encodedText
IF bit IS "0"
MOVE TO root left child
ELSE
MOVE TO root right child
IF current node IS a leaf node
APPEND current node character TO decodedText
MOVE TO root node
RETURN decodedText

END ALGORITHM
```

Sample Input:
text = "hello huffman"

Expected Output:
110 1010 00 00 1110 1011 110 1111 01 01 1000 1001 001

"hello huffman"

Time Complexity: The time complexity for building the Huffman tree is O(nlog⁡n)O(n \log n).

---

3. एक प्रोग्राम लिखें जो डायनेमिक प्रोग्रामिंग का उपयोग करके ऑल पेयर शॉर्टेस्ट पाथ्स को इंप्लीमेंट करे।
Write a program to implement all pair shortest paths using dynamic programming.

ALGORITHM Floyd_Warshall(graph, n)
INPUT: A weighted graph represented as an adjacency matrix, graph, and the number of vertices, n
OUTPUT: The shortest path distances between all pairs of vertices

// Step 1: Initialize the distance matrix
CREATE a distance matrix, dist, as a copy of the input graph
// Step 2: Compute the shortest path distances
FOR k = 0 TO n-1
FOR i = 0 TO n-1
FOR j = 0 TO n-1
IF dist[i][k] ≠ INF AND dist[k][j] ≠ INF
dist[i][j] = MIN(dist[i][j], dist[i][k] + dist[k][j])
// Step 3: Print the shortest distance matrix
PRINT "All-Pairs Shortest Paths Matrix:"
FOR i = 0 TO n-1
FOR j = 0 TO n-1
IF dist[i][j] = INF
PRINT "INF "
ELSE
PRINT dist[i][j] " "
PRINT newline
END ALGORITHM

ALGORITHM

INPUT: None
OUTPUT: The shortest path distances between all pairs of vertices in the given graph
// Step 1: Define the graph and number of vertices
DEFINE n = 4
DEFINE graph as an adjacency matrix representing the graph
// Step 2: Call the Floyd-Warshall algorithm
CALL Floyd_Warshall(graph, n)
END ALGORITHM

Sample Input:
graph = [
[0, 3, INF, 7],
[8, 0, 2, INF],
[5, INF, 0, 1],
[2, INF, INF, 0]
]
n = 4, `INF` represents an unreachable path (infinity).

Expected Output:
All-Pairs Shortest Paths Matrix:
0 3 5 6
5 0 2 3
3 6 0 1
2 5 7 0

Time Complexity: The time complexity of the Floyd-Warshall algorithm is O(V^3).

Lab 10

1. डायनेमिक प्रोग्रामिंग का उपयोग करके मल्टी स्टेज ग्राफ़ समस्या को लागू करने के लिए एक प्रोग्राम लिखें।

Write a program to implement Multi stage graph problem using dynamic programming

ALGORITHM
Procedure: FGRAPH (Forward Approach)
Input:
E: Set of directed edges ⟨i, j⟩ with cost c(i, j)
k: Number of stages in the graph
n: Total number of vertices
c(i, j): Cost for each edge ⟨i, j⟩

Output:
P[1..k]: Minimum cost path from vertex 1 to n
COST[1..n]: Minimum cost from each vertex to destination n
Algorithm

```
Procedure FGRAPH(E, k, n, P)
   real COST[n]
   integer D[n - 1], P[k]
   COST[n] ← 0
   for j ← n - 1 to 1 by -1 do
      let r be a vertex such that <j, r> ∈ E and c(j, r) + COST[r] is minimum
      COST[j] ← c(j, r) + COST[r]
      D[j] ← r
   P[1] ← 1
   P[k] ← n
   for j ← 2 to k - 1 do
      P[j] ← D[P[j - 1]]
End FGRAPHlgorithm
```

Procedure: BGRAPH (Backward Approach)
Input:
E: Set of directed edges ⟨i, j⟩ with cost c(i, j)
k: Number of stages in the graph
n: Total number of vertices
c(i, j): Cost for each edge ⟨i, j⟩

Output:
P[1..k]: Minimum cost path from vertex 1 to n
BCOST[1..n]: Minimum cost from source to each vertex

Algorithm:

```
Procedure BGRAPH(E, k, n, P)
   real BCOST[n]
   integer D[n - 1], P[k]
   BCOST[1] ← 0
   for j ← 2 to n do
      let r be a vertex such that <r, j> ∈ E and BCOST[r] + c(r, j) is minimum
      BCOST[j] ← BCOST[r] + c(r, j)
      D[j] ← r
   P[1] ← 1
   P[k] ← n
   for j ← k - 1 to 2 by -1 do
      P[j] ← D[P[j + 1]]
End BGRAPH
```

Sample Input:
Let the graph have 4 stages and 8 vertices.
Edges with costs
(1, 2) = 2, (1, 3) = 1
(2, 4) = 2, (2, 5) = 3

(3, 5) = 2, (3, 6) = 4
(4, 7) = 1, (4, 8) = 3
(5, 7) = 3, (5, 8) = 2
(6, 8) = 2

Time Complexity:
Let n = number of vertices, e = number of edges, k = number of stages
Time Complexity: Cost computation: O(e) Path reconstruction: O(k) Total: O(e + k)

Lab 11

1. डायनेमिक प्रोग्रामिंग का उपयोग करके मैट्रिक्स श्रृंखला गुणन को लागू करने के लिए एक प्रोग्राम लिखें। प्रोग्राम को मैट्रिक्स की दी गई श्रृंखला के उत्पाद की गणना करने के लिए आवश्यक स्केलर गुणन की न्यूनतम संख्या की गणना करनी चाहिए और इस न्यूनतम को प्राप्त करने के लिए मैट्रिक्स के इष्टतम कोष्ठक को प्रिंट करना चाहिए।

Write a program to implement matrix chain multiplication using dynamic programming. The program should calculate the minimum number of scalar multiplications needed to compute the product of a given chain of matrices and print the optimal parenthesization of the matrices to achieve this minimum.

ALGORITHM

Input:

p[0..n]: An array of dimensions such that matrix A[i] has dimensions p[i-1] × p[i]

(i.e., we have n matrices to multiply)

Output:

m[i][j]: Minimum number of scalar multiplications needed to compute A[i]...A[j]

s[i][j]: Index k at which to split the product for optimal cost

MATRIX-CHAIN-ORDER(p)

```
n ← length(p) - 1
for i ← 1 to n do
    m[i][i] ← 0
for l ← 2 to n do        // l is the chain length
    for i ← 1 to n - l + 1 do
        j ← i + l - 1
        m[i][j] ← ∞
        for k ← i to j - 1 do
```

```
        q ← m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]
        if q < m[i][j] then
            m[i][j] ← q
            s[i][j] ← k
    return m and s
```

Example:

If p = [30, 35, 15, 5, 10, 20, 25]

Then we have 6 matrices:

A1: 30×35 A2: 35×15 A3: 15×5 A4: 5×10 A5: 10×20 A6: 20×25

Time Complexity:

Time: $O(n^3)$ Space: $O(n^2)$

2. एन वें कैटलन संख्या की गणना करने के लिए एक प्रोग्राम लिखें।
Write a program to compute the nth Catalan number.

Algorithm

Input: An integer n (to compute the nth Catalan number)
Output: The nth Catalan number

```
def catalan_number(n):
    catalan = [0] * (n + 1)
    catalan[0], catalan[1] = 1, 1
    for i in range(2, n + 1):
        catalan[i] = sum(catalan[j] * catalan[i - j - 1] for j in range(i))
    return catalan[n]
if __name__ == "__main__":
    n = int(input("Enter the value of n: "))
    print(f"The {n}th Catalan number is: {catalan_number(n)}"
```

Sample Input: Enter the value of n: 5
Sample Output: The 5th Catalan number is: 42

Time Complexity: O(n^2) **Space Complexity:** O(n)

1. बैक ट्रैकिंग समस्या के योग को लागू करने के लिए एक प्रोग्राम लिखें।
Write a program to implement the sum of the subset problem using backtracking.

Input:
A list of numbers (set)
A target sum value

Output:
All combinations (subsets) of numbers from the set that add up to the target
Algorithm

```
def sum_of_subsets(set_list, target):
    n = len(set_list)
    set_list.sort()  # optional, helps to prune branches early

    def backtrack(index, current_sum, subset):
        if current_sum == target:
            print("Solution:", subset)
            return
        if index == n or current_sum > target:
            return
        # Include current element
        subset.append(set_list[index])
        backtrack(index + 1, current_sum + set_list[index], subset)
        subset.pop()  # backtrack
        # Exclude current element
        backtrack(index + 1, current_sum, subset)
    backtrack(0, 0, [])
set_list = [10, 7, 5, 18, 12, 20, 15]
target = 35
sum_of_subsets(set_list, target)
```

Sample Input:
set = [10, 7, 5, 18, 12, 20, 15]
target = 35

Sample Output:
Solution: [10, 5, 20]
Solution: [10, 7, 18]
Solution: [5, 12, 18]

Time Complexity:
Worst case is O(2^n)

---

2. बैकट्रैकिंग का उपयोग करके ग्राफ़ रंग समस्या को लागू करने के लिए एक प्रोग्राम लिखें।

Write a program to implement a graph coloring problem using backtracking.

ALGORITHM : Graph Coloring using Backtracking

Input:
graph – adjacency list representation of the graph
N – number of vertices
M – number of colors

Output:
A valid color assignment for each vertex such that no two adjacent vertices have the same color

```
def is_safe(graph, vertex, color, colors):
    for neighbor in graph[vertex]:
        if colors[neighbor] == color:
            return False
    return True
def backtrack(graph, vertex, colors, M, N):
    if vertex == N:
        print("Solution:", colors)
        return True
    for color in range(1, M + 1):
        if is_safe(graph, vertex, color, colors):
            colors[vertex] = color
            if backtrack(graph, vertex + 1, colors, M, N):
                return True
            colors[vertex] = -1  # backtrack
    return False
def graph_coloring(graph, N, M):
    colors = [-1] * N
    if not backtrack(graph, 0, colors, M, N):
        print("No solution exists")
```

Sample Input:

```
graph = {
0: [1, 2],
1: [0, 2],
2: [0, 1]
}
N = 3
```

M = 3
graph_coloring(graph, N, M)

Sample Output:
Solution: [1, 2, 3]

Time Complexity:
Worst case is O(M^N), where N is number of vertices and M is number of colors

---

3. बैकट्रैकिंग का उपयोग करके एन-क्वींस समस्या को हल करने के लिए एक प्रोग्राम लिखें।
Write a program to solve the N-Queens problem using backtracking.

Input: N – Number of queens (also size of the N x N chessboard)

Output:
One valid arrangement of N queens such that no two queens attack each other.
If no solution exists, display an appropriate message.
Algorithm: Solve N-Queens Problem using Backtracking

```
def is_safe(board, row, col, N):
    # Check column
    for i in range(row):
        if board[i][col] == 1:
            return False
    # Check upper-left diagonal
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1
    # Check upper-right diagonal
    i, j = row - 1, col + 1
    while i >= 0 and j < N:
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1
    return True
def backtrack(row, board, N):
    if row == N:
        print("Solution:")
```

```
        for r in board:
            print(r)
        return True
    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = 1
            if backtrack(row + 1, board, N):
                return True
            board[row][col] = 0  # backtrack
    return False
def solve_n_queens(N):
    board = [[0 for _ in range(N)] for _ in range(N)]
    if not backtrack(0, board, N):
        print("No solution exists")
```

Sample Input:

```
N = 4
solve_n_queens(N)
```

Sample Output:

```
Solution:
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
```

Time Complexity:
Worst case is O(N!) due to all possible queen placements

Lab 13

1. 15-पज़ल समस्या को लीस्ट कॉस्ट ब्रांच एंड बाउंड तकनीक का उपयोग करके हल करने के लिए एक प्रोग्राम लिखें।

   Write a program to implement 15 puzzle problems using the least cost branch and bound

   ALGORITHM: Solve15Puzzle using Least Cost Branch and Bound (LCBnB).
   Input: start_state – Initial configuration of the 15-puzzle as a list or grid (4x4)
   Output:
   Path from start_state to goal_state (if possible), otherwise a message "No solution exists"
   Input: Initial state of the puzzle (initial), Goal state (goal)

Output: Solution path from initial to goal state or failure if no solution

Define a Node structure:
- puzzle_state
- level (depth of the node in the tree)
- cost (calculated using heuristic: f(n) = g(n) + h(n))
- parent (pointer to the parent node)

Define a priority queue PQ (min-heap based on cost f(n))
Calculate the heuristic function h(n):
- h(n) = number of misplaced tiles OR Manhattan distance for each tile

Create the root node:
- state = initial
- level = 0
- cost = h(initial)
- parent = None

Insert root node into PQ
While PQ is not empty:
a. Pop the node N with the least cost from PQ
b. If N.puzzle_state == goal:
- Backtrack from N to root to get solution path
- Return the solution path
c. Generate all valid child states from N by sliding a tile:
- For each child_state:
i. Create a new node:
- state = child_state
- level = N.level + 1
- cost = level + h(child_state)
- parent = N
ii. Add the new node to PQ

If goal state is not found and PQ is empty:
- Return "No solution"

Sample Description:
start_state =
[[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12],
[13, 15, 14, 0]]

goal_state =
[[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12],
[13, 14, 15, 0]]

Time Complexity:
In the worst case, the algorithm behaves like Uniform Cost Search (if no heuristic is used), and explores all nodes up to a certain depth.
Time Complexity: O(b^d)

Where:
b = branching factor (max 4 in 15-puzzle)
d = depth of solution
So, O(4^d) in the worst case

---

2. एक प्रोग्राम लिखें जो ब्रांच एंड बाउंड का उपयोग करके **0/1** नैपसैक प्रॉब्लम को इंप्लीमेंट करे।

Write a program to implement 0/1 knapsack problem using branch and bound.

ALGORITHM
Input:
weights[]: An array of weights of n items
values[]: An array of corresponding values of n items
capacity: The maximum weight capacity of the knapsack
n: Number of items
Output:
max_value: The maximum total value that can be placed in the knapsack without exceeding the capacity.
Define a Node structure:
- level: index of the current item
- profit: total value so far
- weight: total weight so far
- bound: upper bound of maximum profit from this node
- items_included: list of included item indices

Define a priority queue PQ (max-heap based on bound)
Sort all items by value/weight ratio in descending order
Define a function `bound(node)`:
- If node.weight ≥ W: return 0
- Initialize result = node.profit
- Set j = node.level + 1, total_weight = node.weight
- While j < n and total_weight + weights[j] ≤ W:
  - total_weight += weights[j]
  - result += values[j]
  - j += 1
- If j < n:
  - result += (W - total_weight) * (values[j] / weights[j])
- Return result

Create a root node:
- level = -1
- profit = 0
- weight = 0
- bound = bound(root)
- items_included = []

Insert root node into PQ
max_profit = 0
While PQ is not empty:

Remove node u with highest bound from PQ
If u.bound > max_profit:
i. Create node v (left child = taking next item)
- level = u.level + 1
- weight = u.weight + weights[v.level]
- profit = u.profit + values[v.level]
- items_included = u.items_included + [v.level]
- If weight ≤ W and profit > max_profit:
  - max_profit = profit
- bound = bound(v)
- If bound > max_profit:
  - Add v to PQ

ii. Create node v (right child = not taking next item)
- level = u.level + 1
- weight = u.weight
- profit = u.profit
- items_included = u.items_included
- bound = bound(v)
- If bound > max_profit:
  - Add v to PQ

9. Return max_profit

Sample Input:
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5 n = 4
Sample Output: Maximum value in knapsack = 7
Time Complexity:
Worst Case: O(2^n)
Optimized Case (using bounding): O(n log n)

---

3. ब्रांच और बाउंड का उपयोग करके ट्रैवलिंग सेल्समैन समस्या को इंप्लीमेंट करने के लिए एक प्रोग्राम लिखें।

Write a program to implement a travelling salesman problem using branch and bound.

ALGORITHM
Input:
cost_matrix[n][n]: A 2D matrix where cost_matrix[i][j] is the cost of traveling from city i to city j
n: Total number of cities
Output:
min_cost: The minimum cost to complete the tour
best_path: The sequence of cities representing the best tour

Define a Node structure:
- level: depth in the tree (number of cities visited)

- path: list of visited cities in order
- reduced_matrix: cost matrix after row and column reduction
- cost: total cost so far (current path cost + lower bound)
- vertex: current city

Use a priority queue PQ (min-heap) ordered by node.cost
Define a function reduceMatrix(matrix):
- For each row, subtract the minimum element (if not ∞)
- For each column, subtract the minimum element (if not ∞)
- Return the reduced matrix and the total reduction cost

Define a function createChildNode(parent, i, matrix):
- Copy parent's reduced matrix
- Set all entries in parent.vertex row and column i to ∞
- Set matrix[i][0] to ∞ (to avoid premature return to start)
- Apply reduceMatrix on updated matrix
- Calculate new cost = parent.cost + cost from parent.vertex to i + reduction cost
- Return the new node

Initialize root node:
- path = [0] (start at city 0)
- reduced_matrix, reduction_cost = reduceMatrix(cost_matrix)
- cost = reduction_cost
- level = 0
- vertex = 0

Insert root node into PQ
Initialize:
- min_cost = ∞
- best_path = []

While PQ is not empty:
- Pop node with the lowest cost from PQ (current_node)
- If current_node.level == N - 1:
  - Complete the path by returning to the start
  - total_cost = current_node.cost + cost_matrix[current_node.vertex][0]
  - If total_cost < min_cost:
    - min_cost = total_cost
    - best_path = current_node.path + [0]
- Else:
  - For each city i not in current_node.path:
    - child_node = createChildNode(current_node, i, current_node.reduced_matrix)
    - child_node.path = current_node.path + [i]
    - child_node.level = current_node.level + 1
    - child_node.vertex = i
    - Insert child_node into PQ

Return min_cost and best_path

Sample Input:

```
cost_matrix = [
   [inf, 10, 15, 20],
   [10, inf, 35, 25],
   [15, 35, inf, 30],
   [20, 25, 30, inf]
```

]
n = 4
Sample Output:
Minimum cost: 80
Best path: [0, 1, 3, 2, 0]

Time Complexity:

Worst Case: O(n!) Optimized with Branch and Bound: $O(n^2 \times 2^n)$

विषय समन्वयक (Subject Coordinators)

डॉ मानसी ज्ञानचंदानी (Dr. Manasi Gyanchandani)

डॉ. श्वेता जैन (Dr. Sweta Jain)

डॉ. प्रगति अग्रवाल (Dr. Pragati Agrawal)