Algorithm for Linear Search (practical\_4(a)):

Define a function LinearSearch(array, n, k) with parameters array, n, and k.

Loop through the array using the range from 0 to n.

Check if the current element at index j is equal to k.

If it is, return the index j.

If the element is not found in the loop, return -1.

Define the array to be searched.

Define the length of the array and the element to search for.

Call the LinearSearch function with the defined parameters and store the result in the variable result.

Check if the result is equal to -1.

If it is, print "Element not found".

Else, print "Element found at index: " and the value of the result.

Algorithm for Binary search (iterative) (practical\_4(b)):

Define a function binarySearch(arr, k, low, high) with parameters arr, k, low, and high.

Use a while loop to perform binary search on the array.

Calculate the middle index mid as the average of low and high indices.

Check if the element at the mid index is equal to k.

If it is, return the mid index.

If the element at mid is less than k, set low to mid + 1.

If the element at mid is greater than k, set high to mid - 1.

If the element is not found in the loop, return -1.

Define the array to be searched, the element to search for, and the initial low and high indices.

Call the binarySearch function with the defined parameters and store the result in the variable result.

Check if the result is not equal to -1.

If it is not, print "Element is present at index " and the value of the result.

Else, print "Not found".

Algorithm for Binary search (recursive) (practical\_4(b)):

Define a function BinarySearch(arr, k, low, high) with parameters arr, k, low, and high.

Check if high is greater than or equal to low.

Calculate the middle index mid as the average of low and high indices.

Check if the element at the mid index is equal to k.

If it is, return the mid index.

If the element at mid is greater than k, call the BinarySearch function recursively with the parameters arr, k, low, and mid-1.

If the element at mid is less than k, call the BinarySearch function recursively with the parameters arr, k, mid+1, and high.

If the element is not found in the loop, return -1.

Define the array to be searched, the element to search for, and the initial low and high indices.

Call the BinarySearch function with the defined parameters and store the result in the variable result.

Check if the result is not equal to -1.

If it is not, print "Element is present at index " and the value of the result.

Else, print "Not found".

Practical\_5

Algorithm for Bubble Sort:

Get the input array and find its length n.

Loop through the array from i = 0 to n-1.

Within the above loop, loop again through the array from j = 0 to n-i-1.

Compare arr[j] with arr[j+1]. If arr[j] is greater than arr[j+1], swap the two elements.

After the above inner loop, the last i elements of the array will be sorted.

Repeat steps 2-5 until the entire array is sorted.

Print the sorted array.

Algorithm for Selection Sort:

Get the input array A and find its length n.

Loop through the array from i = 0 to n-1.

Within the above loop, set min\_idx to i.

Loop again through the array from j = i+1 to n-1.

If A[j] is less than A[min\_idx], set min\_idx to j.

Swap A[i] with A[min\_idx].

Repeat steps 2-6 until the entire array is sorted.

Print the sorted array.

Algorithm for Insertion Sort:

Get the input array arr and find its length n.

Loop through the array from i = 1 to n-1.

Within the above loop, set key to arr[i].

Loop through the array from j = i-1 to 0, decrementing j each time.

If arr[j] is greater than key, move arr[j] to arr[j+1].

Repeat step 4 until arr[j] is less than or equal to key.

Set arr[j+1] to key.

Repeat steps 2-7 until the entire array is sorted.

Print the sorted array.

Practical\_6

Algorithm for finding the minimum element in a list:

Initialize a list lst with elements

Sort the list in ascending order using the sort() method

Print the first element of the sorted list using indexing to get the minimum element.

Pseudocode:

lua

Copy code

function minimum():

lst = [10, 5, 75, 36, 86, 54, 6, 7, 65, 12]

lst.sort()

print("Minimum element is: ", lst[0])

Algorithm for finding the maximum element in a list:

Initialize a list lst with elements

Sort the list in ascending order using the sort() method

Print the last element of the sorted list using negative indexing to get the maximum element.

Pseudocode:

lua

Copy code

function maximum():

lst = [10, 5, 75, 36, 86, 54, 6, 7, 65, 12]

lst.sort()

print("Maximum element is: ", lst[-1])

practical\_7

Start by defining the function brute\_force\_search that takes in two parameters pattern and string.

In the function, loop through the indices of string up to len(string) - len(pattern) + 1 using a for loop with i as the loop variable.

Initialize j to 0.

Use a while loop to compare the characters in pattern to the corresponding characters in string starting at index i and continuing while j is less than the length of pattern and the characters match.

If j equals the length of pattern, return the index i.

If the while loop terminates without finding a match, increment i and repeat the process.

If no match is found after looping through all the indices, return -1.

Call the brute\_force\_search function with the pattern and string variables.

Print the returned index from the brute\_force\_search function.

Practical\_8

The factorial function takes an integer n as input and returns the factorial of n. If n is equal to 0, the function returns 1; otherwise, it returns n multiplied by the factorial of n-1.

The fibonacci function takes an integer n as input and returns the nth Fibonacci number. If n is less than or equal to 1, the function returns n. Otherwise, it returns the sum of the (n-1)th and (n-2)th Fibonacci numbers.

The tower\_of\_hanoi function solves the classic Tower of Hanoi puzzle, which involves moving a stack of disks from one pole to another, subject to certain constraints. The function takes four inputs: the number of disks (n), the name of the source pole (source), the name of the destination pole (destination), and the name of the auxiliary pole (auxiliary). If n is equal to 1, the function prints a message indicating that the top disk should be moved from the source pole to the destination pole. Otherwise, the function calls itself recursively with n-1 disks, moving them from the source pole to the auxiliary pole. It then prints a message indicating that the nth disk should be moved from the source pole to the destination pole. Finally, the function calls itself recursively with n-1 disks, moving them from the auxiliary pole to the destination pole.

The iterative\_factorial function takes an integer n as input and returns the factorial of n using an iterative approach. It initializes result to 1 and then multiplies it by the integers from 1 to n using a for loop. It returns result.

The iterative\_fibonacci function takes an integer n as input and returns the nth Fibonacci number using an iterative approach. If n is less than or equal to 1, the function returns n. Otherwise, it initializes fib\_0 to 0 and fib\_1 to 1, and then uses a for loop to compute the nth Fibonacci number by repeatedly adding fib\_0 and fib\_1 and updating their values. Finally, the function returns the value of fib\_1.

Practical\_9

Start by importing the necessary libraries, such as NumPy.

Define the Strassen algorithm function that takes two matrices, x and y, as inputs.

If the size of either x or y is equal to 1, return the product of x and y.

Get the size of the input matrices.

If the size of x is odd, pad it with an additional row and column of zeros.

Calculate the value of m as the ceil of n/2.

Divide the matrices into 4 submatrices each, namely a, b, c, d, e, f, g, h.

Calculate the values of the 7 partial products p1 to p7, using the submatrices.

Create a new result matrix of size 2m x 2m, and calculate its values based on the partial products.

Return the result matrix with its size reduced to n x n.

Define the main function.

Create two matrices x and y, and call the strassen\_algorithm function with these matrices as inputs.

Print the resulting matrix.

Practical\_10

Algorithm for the lcs function:

Inputs:

s1: first string

s2: second string

Outputs:

Length of the longest common subsequence between s1 and s2

Initialize m to be the length of s1, n to be the length of s2, and lcs to be a 2D array of size (m+1) x (n+1) filled with zeros

For i from 1 to m+1:

For j from 1 to n+1:

If the i-1th character of s1 is equal to the j-1th character of s2:

Set lcs[i][j] to lcs[i-1][j-1] + 1

Otherwise:

Set lcs[i][j] to max(lcs[i-1][j], lcs[i][j-1])

Return lcs[m][n]