Priyanshu Sharma
CS24042

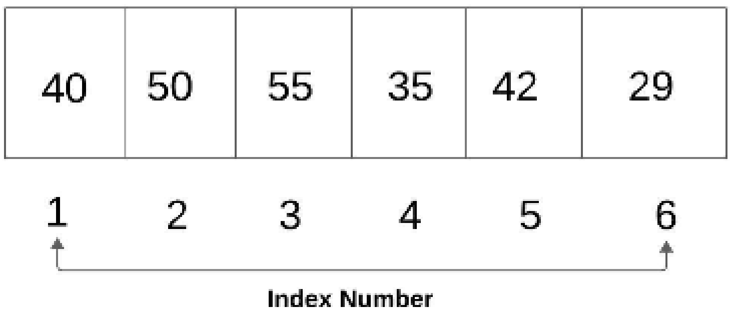
Practical No 1

Aim:- Array Operations: Implement programs for 1-d arrays, Implement programs for 2-d arrays.

Definition:

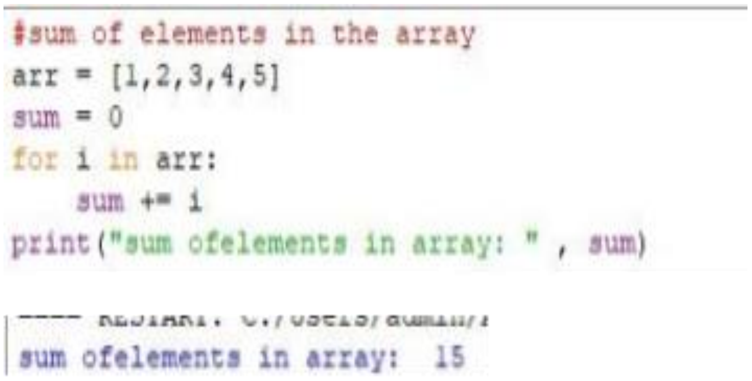
1-d arrays: A **1-Dimensional (1-D) Array** is a collection of elements that are stored in a linear order. It is a list of values of the same data type, where each element can be accessed by its index. The array size is fixed, and the elements are stored in consecutive memory locations.

Diagram:



Programs on 1-d arrays like:

1. sum of elements of array

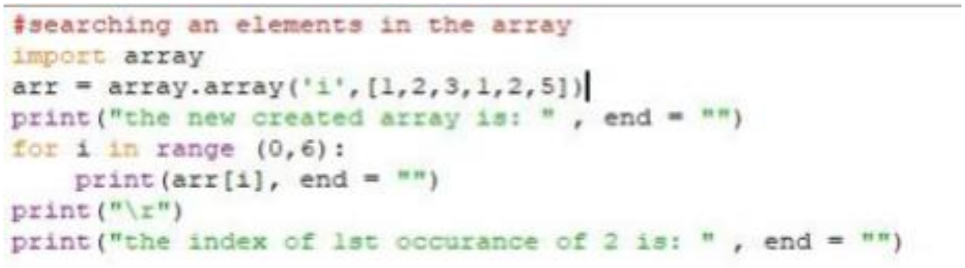


```
#sum of elements in the array
arr = [1,2,3,4,5]
sum = 0
for i in arr:
    sum += i
print("sum ofelements in array: " , sum)
```

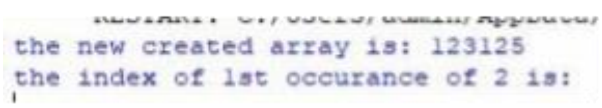
```
sum ofelements in array:  15
```

Priyanshu Sharma
CS24042

2. searching an element in array



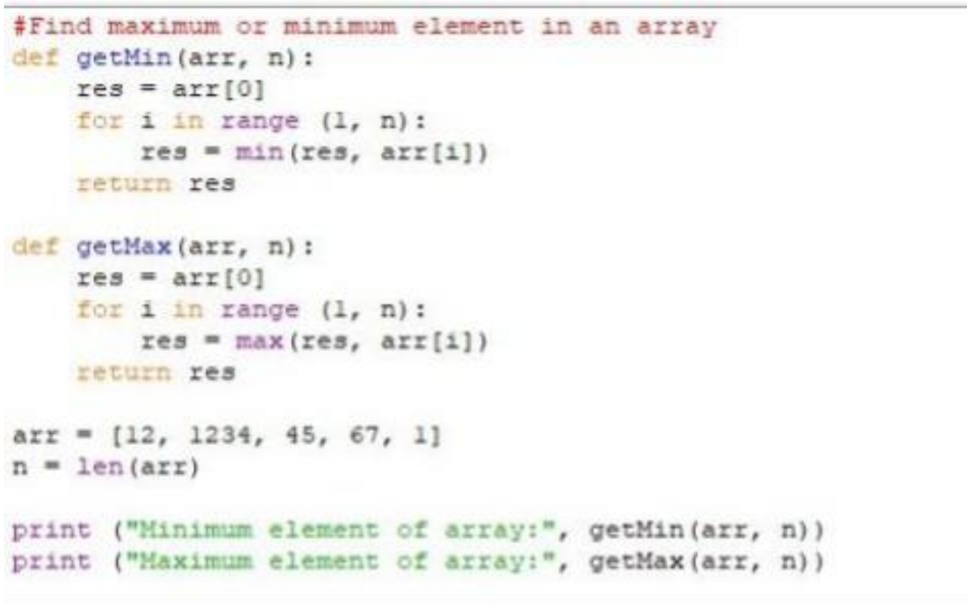
```
#searching an elements in the array
import array
arr = array.array('i',[1,2,3,1,2,5])
print("the new created array is: " , end = "")
for i in range (0,6):
    print(arr[i], end = "")
print("\r")
print("the index of 1st occurance of 2 is: " , end = "")
```



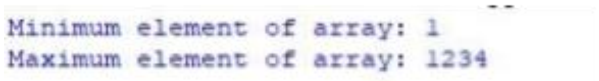
```
the new created array is: 123125
the index of 1st occurance of 2 is:
```

Priyanshu Sharma
CS24042

3. finding minimum and maximum element in array

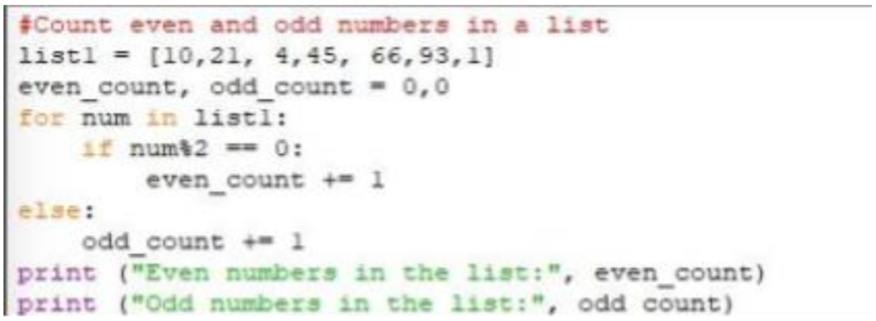


```
#Find maximum or minimum element in an array
def getMin(arr, n):
    res = arr[0]
    for i in range (1, n):
        res = min(res, arr[i])
    return res

def getMax(arr, n):
    res = arr[0]
    for i in range (1, n):
        res = max(res, arr[i])
    return res

arr = [12, 1234, 45, 67, 1]
n = len(arr)

print ("Minimum element of array:", getMin(arr, n))
print ("Maximum element of array:", getMax(arr, n))
```

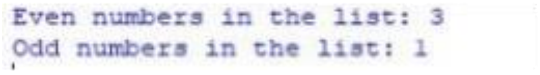


```
Minimum element of array: 1
Maximum element of array: 1234
```

4. count the number of even and odd numbers in array.



```
#Count even and odd numbers in a list
list1 = [10,21, 4,45, 66,93,1]
even_count, odd_count = 0,0
for num in list1:
    if num%2 == 0:
        even_count += 1
else:
    odd_count += 1
print ("Even numbers in the list:", even_count)
print ("Odd numbers in the list:", odd count)
```

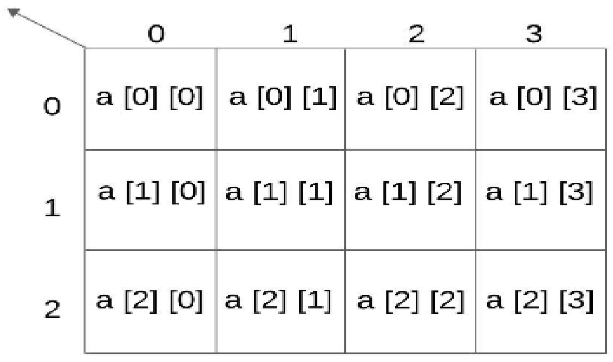


```
Even numbers in the list: 3
Odd numbers in the list: 1
```

Priyanshu Sharma
CS24042

Definition -2-d arrays: A 2-Dimensional (2-D) Array is a collection of elements organized in a matrix form, where data is stored in rows and columns. It is essentially an array of arrays. Each element is accessed by two indices: one for the row and one for the column.

Diagram:

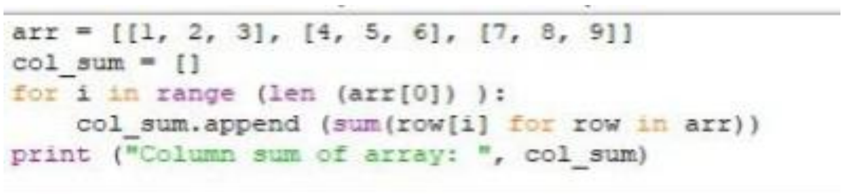


	0	1	2	3
0	a [0] [0]	a [0] [1]	a [0] [2]	a [0] [3]
1	a [1] [0]	a [1] [1]	a [1] [2]	a [1] [3]
2	a [2] [0]	a [2] [1]	a [2] [2]	a [2] [3]

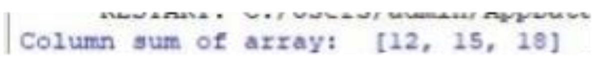
Find a [1] [3] ?

Programs on 2-d arrays like:

1. row-sum, column-sum

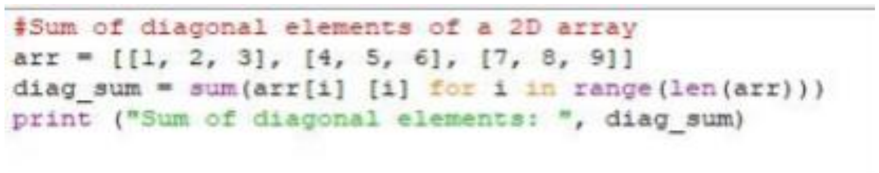


```
arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
col_sum = []
for i in range (len (arr[0]) ):
    col_sum.append (sum(row[i] for row in arr))
print ("Column sum of array: ", col_sum)
```

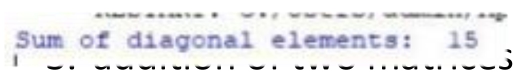


```
Column sum of array:  [12, 15, 18]
```

2. sum of diagonal elements



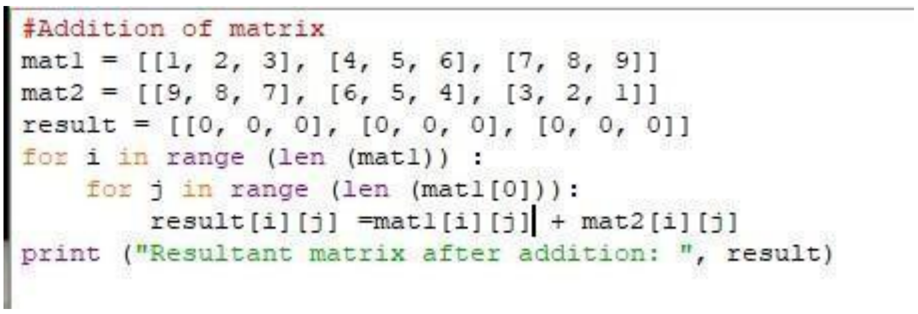
```
#Sum of diagonal elements of a 2D array
arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
diag_sum = sum(arr[i] [i] for i in range(len(arr)))
print ("Sum of diagonal elements: ", diag_sum)
```



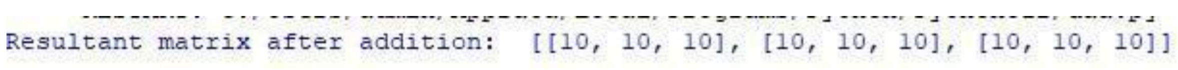
```
Sum of diagonal elements:  15
```

3. addition of two matrices

Priyanshu Sharma
CS24042

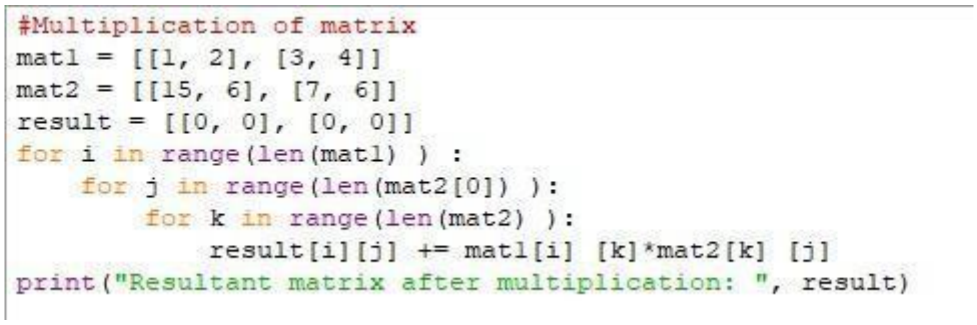


```
#Addition of matrix
mat1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
mat2 = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
result = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
for i in range (len (mat1)) :
    for j in range (len (mat1[0])):
        result[i][j] =mat1[i][j] + mat2[i][j]
print ("Resultant matrix after addition: ", result)
```

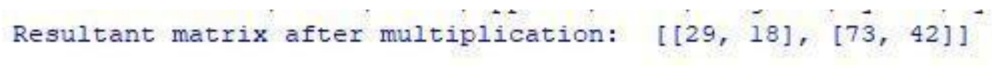


```
Resultant matrix after addition:  [[10, 10, 10], [10, 10, 10], [10, 10, 10]]
```

4. multiplication of two matrices



```
#Multiplication of matrix
mat1 = [[1, 2], [3, 4]]
mat2 = [[15, 6], [7, 6]]
result = [[0, 0], [0, 0]]
for i in range(len(mat1) ) :
    for j in range(len(mat2[0]) ):
        for k in range(len(mat2) ):
            result[i][j] += mat1[i] [k]*mat2[k] [j]
print("Resultant matrix after multiplication: ", result)
```



```
Resultant matrix after multiplication:  [[29, 18], [73, 42]]
```

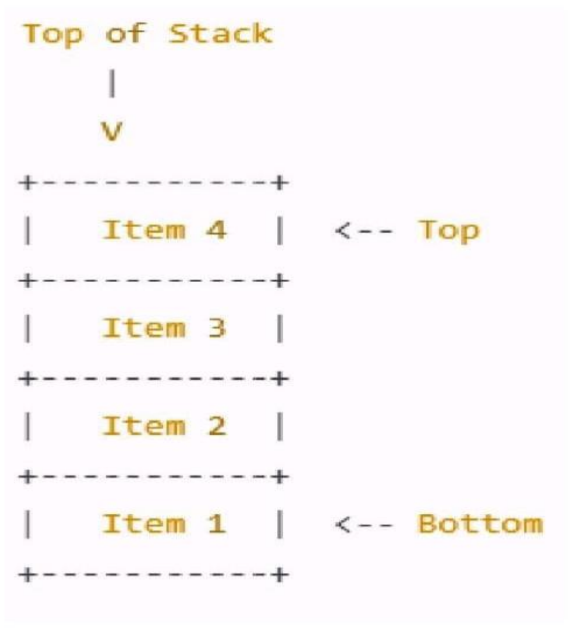
Priyanshu Sharma
CS24042

Practical No 2

Aim:- Create a list-based stack and perform stack operations.

Definition of Stack:

A stack is a data structure that follows the Last In, First Out (LIFO) principle. You can only add (push) or remove (pop) elements from the top. It's like a stack of plates—add a plate to the top and remove the top plate when needed.



Stack Diagram:

List Based Stack Operations (Explain it in one line):

- **Pop**: Removes and returns the top element of the stack.
- **Push**: Adds an element to the top of the stack.
- **Peek**: Returns the top element without removing it.
- **Is_empty**: Checks if the stack is empty.
- **Size**: Returns the number of elements in the stack.

Source Code:

```
class Stack:

    def _init_(self):
        self.items = []

    def push(self, item):
```

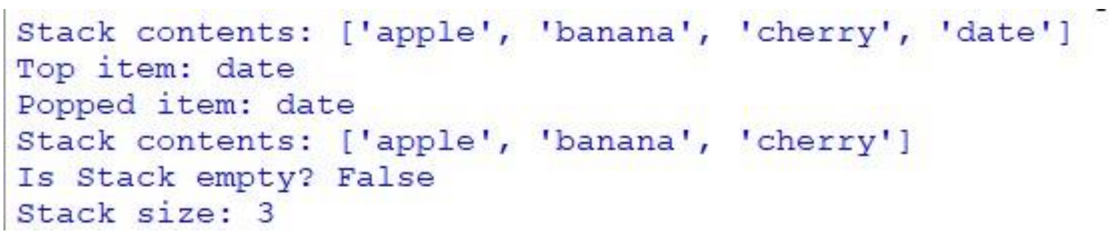
Priyanshu Sharma
CS24042

```
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def is_empty(self):
        return len(self.items)==0

    def size(self):
        return len(self.items)

#create a new stack object
s=Stack()

#push some items onto the stack
s.push('apple')
s.push('banana')
s.push('cherry')
s.push('date')

# print the curent state of the stack
print("Stack contents:",s.items)

# peek at the top item of the stack
print("Top item:",s.peek())

#pop an item off the stack
popped_item = s.pop()
print("Popped item:",popped_item)
```

Priyanshu Sharma
CS24042

```
#print the current state of the stack

print("Stack contents:",s.items)


#check if the stack is empty

print("Is Stack empty?",s.is_empty())


#get the size of the stack

print("Stack size:",s.size())
```

Output:



```
Stack contents: ['apple', 'banana', 'cherry', 'date']
Top item: date
Popped item: date
Stack contents: ['apple', 'banana', 'cherry']
Is Stack empty? False
Stack size: 3
```

Priyanshu Sharma
CS24042

PRACTICAL 3

AIM: Implement Linear and Binary Search Algorithm on a list.
Defination:
Linear Search
Linear search is a simple searching algorithm that checks each element in a list sequentially until the desired element is found or the list ends. It is best suited for small or unsorted datasets.

- Time Complexity: O(n)O(n)O(n) in the worst case
- Best Case: O(1)O(1)O(1) (if the element is at the beginning)
- Worst Case: O(n)O(n)O(n) (if the element is at the end or not present)

Binary Search
Binary search is an efficient searching algorithm that works on sorted lists. It repeatedly divides the search space in half, comparing the target value with the middle element and eliminating half of the list each time.

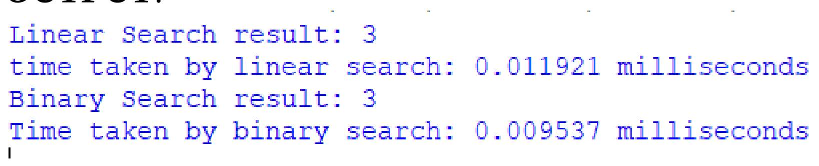
- Time Complexity: O(log⁡n)O(\log n)O(logn)
- Best Case: O(1)O(1)O(1) (if the middle element is the target)
- Worst Case: O(log⁡n)O(\log n)O(logn)

Code:
```
import time
def linear_search(arr,x):
   for i in range(len(arr)):
      if arr[i]==x:
         return i
   return -1
def binary_search(arr,X):
   low = 0
   high = len(arr) - 1
   mid = 0
   while low <= high:
      mid = (high + low) // 2
      if arr[mid] < X:
         low = mid + 1
      elif arr[mid] > X:
         high = mid - 1
      else:
         return mid
   return -1

arr = [2,5,8,10,13,10,22,25,20,30]

start = time.time()
result = linear_search(arr, 10)
end = time.time()
print("Linear Search result:",result)
print("time taken by linear search: {:.6f} milliseconds".format((end - start) * 1000))

start = time.time()
result = binary_search(arr, 10)
end = time.time()
print("Binary Search result:", result)
```

Priyanshu Sharma
CS24042

*print("Time taken by binary search: {:.6f} milliseconds".format((end - start) * 1000))*

***OUTPUT**:*



```
Linear Search result: 3
time taken by linear search: 0.011921 milliseconds
Binary Search result: 3
Time taken by binary search: 0.009537 milliseconds
```

Priyanshu Sharma
CS24042

Practical No 4

Aim:- Implement sorting algorithms (e.g., bubble, selection, insertion).

Explain Bubble Sort: Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the list.

Explain Selection Sort: Selection Sort is another simple sorting algorithm that divides the input list into two parts: a sorted part and an unsorted part. It repeatedly selects the smallest (or largest, depending on the order) element from the unsorted part and moves it to the end of the sorted part.

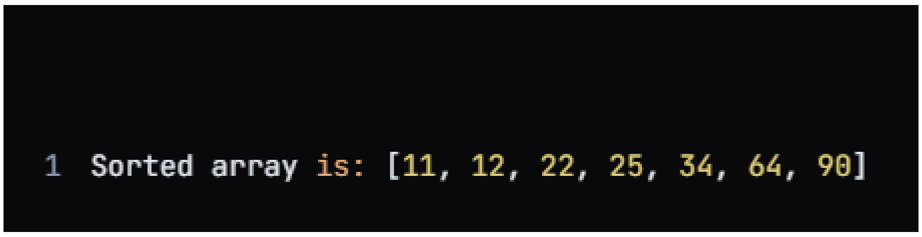
Explain Insertion Sort: Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. The algorithm works similarly to the way you might sort playing cards in your hands.

Program 1: Bubble Sort

Source Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                # Swap elements if they are in the wrong order
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Example usage:
# Initialize an array
my_array = [64, 34, 25, 12, 22, 11, 90]

# Display the original array
print("Original array:", my_array)

# Apply the bubble sort algorithm
bubble_sort(my_array)

# Display the sorted array
print("Sorted array:", my_array)
```

Output:



```
Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

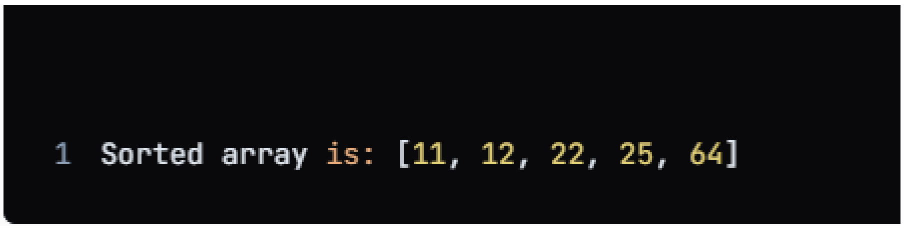
Program 2: Selection Sort

Priyanshu Sharma
CS24042

Source Code:

```python
def selection_sort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        # Find the minimum element in the unsorted part
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        # Swap the found minimum element with the first element
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Example usage:
my_array = [64, 25, 12, 22, 11]
selection_sort(my_array)
print("Sorted array:", my_array)
```

Output:



```
Sorted array is: [11, 12, 22, 25, 64]
```

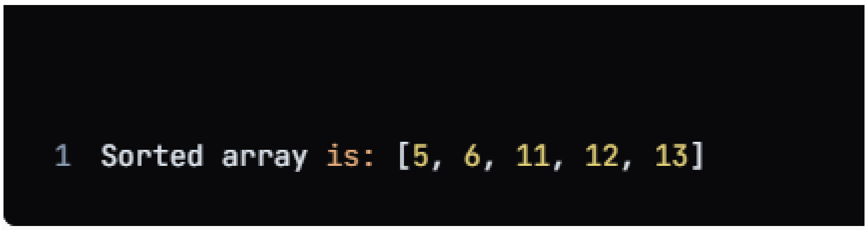
Program 3: Insertion Sort

Source Code:

Priyanshu Sharma
CS24042

```python
def insertion_sort(arr):
    n = len(arr)

    # step1: If the element is the first element,
    #assume that it is already sorted.
    # Return 1 (In Python, we start indexing from 0,
    #so for the first element, it's already "sorted".)
    if n <= 1:
        return arr

    # step2: Iterate through the array starting
    #from the second element
    for i in range(1, n):
        # step2: Pick the next element and
        #store it separately in a key
        key = arr[i]

        # step3: Compare the key with all
        #elements in the sorted array
        j = i - 1

        # step4: If the element in the sorted array is smaller,
        #move to the next element or shift greater elements to the right
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        # step5: Insert the value
        arr[j + 1] = key

    # step6: Repeat until the array is sorted,
    #and return the sorted array
    return arr

# Example usage:
arr = [12, 11, 13, 5, 6]
sorted_arr = insertion_sort(arr)
print("Sorted array:", sorted_arr)
```

Output:



```
Sorted array is: [5, 6, 11, 12, 13]
```

Priyanshu Sharma
CS24042

Practical 5

AIM: Implement algorithms to find Nth Max/Min element in a list.

Definition

Nth Maximum:

The **Nth maximum** of a set or array is the element that ranks as the Nth largest when all the elements are arranged in decreasing order. In other words, it is the element that would occupy the Nth position in a sorted list when sorted from the largest to the smallest.

Nth Minimum:

The **Nth minimum** of a set or array is the element that ranks as the Nth smallest when all the elements are arranged in increasing order. It is the element that would occupy the Nth position in a sorted list when sorted from the smallest to the largest.

CODE:

```
import time import heapq


# Function to find the Nth maximum element using

sorting def nth_max_sort(lst, n):     sorted_list =

sorted(lst, reverse=True)     return sorted_list[n-1]


# Function to find the Nth maximum element using

heapq.nlargest def nth_max_heap(lst, n):

    return heapq.nlargest(n, lst)[-1]


# Function to find the Nth maximum element using max function

iteratively def nth_max_max(lst, n):

    temp_lst = lst.copy()  # Avoid modifying the

original list     for _ in range(n-1):

temp_lst.remove(max(temp_lst))     return

max(temp_lst)


# Function to find the Nth minimum element using
```

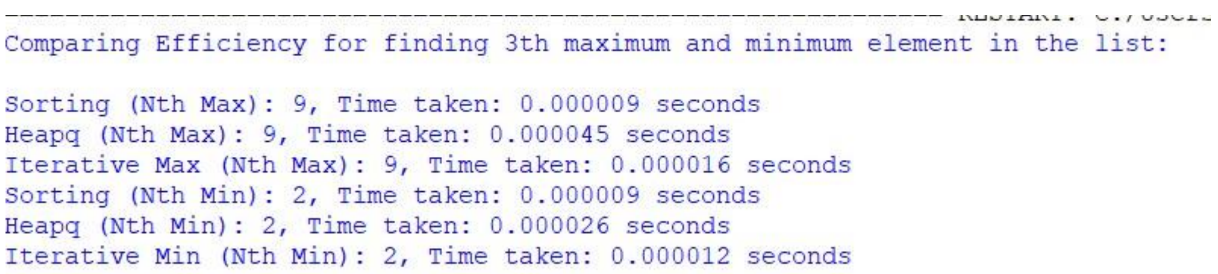
Priyanshu Sharma
CS24042

```
sorting def nth_min_sort(lst, n):     sorted_list =
sorted(lst)     return sorted_list[n-1]


# Function to find the Nth minimum element using
heapq.nsmallest def nth_min_heap(lst, n):
    return heapq.nsmallest(n, lst)[-1]

# Function to find the Nth minimum element using min function
iteratively def nth_min_min(lst, n):
    temp_lst = lst.copy()  # Avoid modifying the
original list     for _ in range(n-1):
temp_lst.remove(min(temp_lst))     return
min(temp_lst)


# Function to compare efficiency of
algorithms def compare_algorithms(lst, n):
    algorithms = {
        "Sorting (Nth Max)": nth_max_sort,
        "Heapq (Nth Max)": nth_max_heap,
        "Iterative Max (Nth Max)": nth_max_max,
        "Sorting (Nth Min)": nth_min_sort,
        "Heapq (Nth Min)": nth_min_heap,
        "Iterative Min (Nth Min)": nth_min_min
    }

    print(f"Comparing Efficiency for finding {n}th maximum and minimum element in the
list:\n")


    for name, func in algorithms.items():
        start_time = time.time()         result = func(lst, n)
elapsed_time = time.time() - start_time         print(f"{name}:
{result}, Time taken: {elapsed_time:.6f} seconds")
```

Priyanshu Sharma
CS24042

```
# Sample list and N value
lst = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3,
8, 4, 6] n = 3

# Run efficiency comparison
compare_algorithms(lst, n)
```

OUTPUT:



```
Comparing Efficiency for finding 3th maximum and minimum element in the list:

Sorting (Nth Max): 9, Time taken: 0.000009 seconds
Heapq (Nth Max): 9, Time taken: 0.000045 seconds
Iterative Max (Nth Max): 9, Time taken: 0.000016 seconds
Sorting (Nth Min): 2, Time taken: 0.000009 seconds
Heapq (Nth Min): 2, Time taken: 0.000026 seconds
Iterative Min (Nth Min): 2, Time taken: 0.000012 seconds
```

Priyanshu Sharma
CS24042

Practical 6

AIM: Implement algorithms to find a pattern in a given string.

Definition

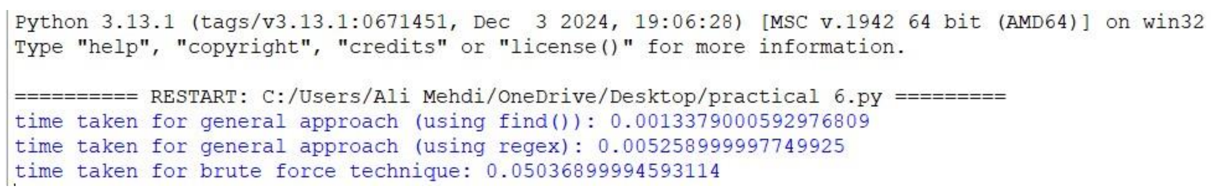
String Pattern Matching is the process of checking whether a string (often called a *pattern*) exists within another string (often called *text*). It involves searching for a specific sequence of characters (the pattern) within a larger sequence of characters (the text). The goal is to find whether the pattern is present in the text, and if it is, return the position or all positions where the pattern occurs.

Best Example of String Pattern Matching: **Knuth-Morris-Pratt (KMP) Algorithm**

CODE:

```
import timeit import re def find_pattern_general(text,pattern):
    return text.find(pattern) != -1 def
find_pattern_regex(text,pattern):     return
bool(re.search(pattern,text)) def
find_pattern_brute_force(text,pattern):
    n=len(text)
m=len(pattern)     for i in
range(n-m+1):
        j=0         while j<m:
if text[i+j] !=pattern[j]:
                break
j+=1         if
j==m:
return True
return False
text = "This is a sample text where we will dearch for a pattern." pattern="search"
Name: Apkesha singh
Roll No: CS24044 t_general_find=timeit.timeit(lambda:
find_pattern_general(text,pattern),number=10000)
t_regex=timeit.timeit(lambda:
find_pattern_regex(text,pattern),number=10000)
t_brute_force=timeit.timeit(lambda:
find_pattern_brute_force(text,pattern),number=10000)
print("time taken for general approach (using find()):", t_general_find) print("time
taken for general approach (using regex):", t_regex) print("time taken for brute force
technique:", t_brute_force)
```

OUTPUT:



```
Python 3.13.1 (tags/v3.13.1:0671451, Dec  3 2024, 19:06:28) [MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

========== RESTART: C:/Users/Ali Mehdi/OneDrive/Desktop/practical 6.py =========
time taken for general approach (using find()): 0.0013379000592976809
time taken for general approach (using regex): 0.005258999997749925
time taken for brute force technique: 0.05036899994593114
```

Priyanshu Sharma
CS24042

Practical 7

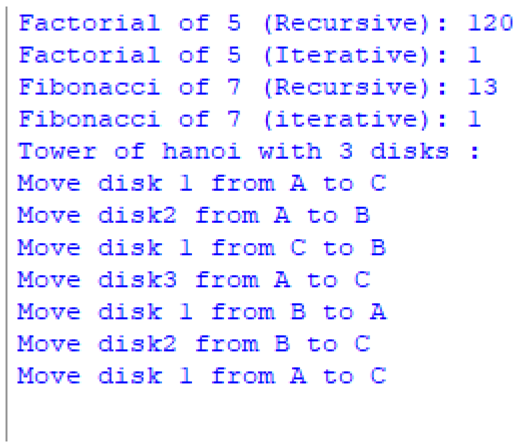
AIM: Implement recursive algorithms (e.g., factorial, Fibonacci, Tower of Hanoi).

Definition of Recursion:
Recursion is a technique in programming where a function calls itself in order to solve a problem.
Code:

```
def factorial_recursive(n):
   if n== 0:
      return 1
   return n * factorial_recursive(n-1)
def factorial_iterative(n):
   result = 1
   for i in range(1, n + 1):
      result *= i
      return result
def fibonacci_recursive(n):
   if n<=1:
      return n
   return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)

def fibonacci_iterative(n):
   if n<= 1:
      return n
   a,b =0,1
   for _ in range(2,n + 1):
      a, b= b, a + b
      return b
def tower_of_hanoi(n, source, target, auxiliary):
   if n==1:
      print(f"Move disk 1 from {source} to {target}")
      return
   tower_of_hanoi (n-1, source, auxiliary, target)
   print(f"Move disk{n} from {source} to {target}")
   tower_of_hanoi (n-1, auxiliary, target, source)

#factorial
print("Factorial of 5 (Recursive):", factorial_recursive(5))
print("Factorial of 5 (Iterative):", factorial_iterative(5))

#Fibonacci
print("Fibonacci of 7 (Recursive):",fibonacci_recursive(7))
print("Fibonacci of 7 (iterative):",fibonacci_iterative(7))

#Tower of Hanoi
print("Tower of hanoi with 3 disks :")
tower_of_hanoi(3,'A','C','B')
```

Output:

Priyanshu Sharma
CS24042



```
Factorial of 5 (Recursive): 120
Factorial of 5 (Iterative): 1
Fibonacci of 7 (Recursive): 13
Fibonacci of 7 (iterative): 1
Tower of hanoi with 3 disks :
Move disk 1 from A to C
Move disk2 from A to B
Move disk 1 from C to B
Move disk3 from A to C
Move disk 1 from B to A
Move disk2 from B to C
Move disk 1 from A to C
```

Priyanshu Sharma
CS24042

Practical 8

AIM: Greedy Algorithm: Solve problems like file merging and coin change using the Greedy Algorithm.

Definition of Greedy Algorithm:
Greedy algorithms work by making the best choice at each step and are particularly useful in optimization problems where this strategy leads to the optimal solution.

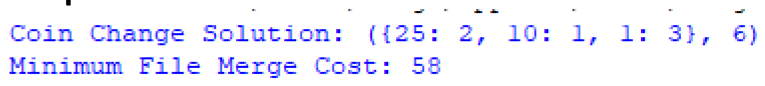
Code:

```
import heapq
# Coin Change Problem (Greedy Approach)
def coin_change_greedy(coins, amount):
    coins.sort(reverse=True)  # Sort coins in descending order
    result = {}
    total_coins = 0

    for coin in coins:
        if amount >= coin:
            count = amount // coin  # Get maximum coins of this denomination
            amount -= count * coin
            result[coin] = count
            total_coins += count

        if amount == 0:
            break

    if amount != 0:
        return "Solution not possible with given denominations"

    return result, total_coins

# File Merging Problem (Greedy Approach using Min-Heap)
def min_file_merge_cost(files):
    heapq.heapify(files)  # Min heap to get smallest files first
    total_cost = 0

    while len(files) > 1:
        # Extract two smallest files
        first = heapq.heappop(files)
        second = heapq.heappop(files)

        merge_cost = first + second
        total_cost += merge_cost

        # Push merged file back into the heap
        heapq.heappush(files, merge_cost)

    return total_cost
```

Priyanshu Sharma
CS24042

```
# Example Usage
if __name__ == "__main__":
    # Coin Change Example
    coins = [1, 5, 10, 25]
    amount = 63
    print("Coin Change Solution:", coin_change_greedy(coins, amount))

    # File Merging Example
    file_sizes = [4, 8, 6, 12]
    print("Minimum File Merge Cost:", min_file_merge_cost(file_sizes))
```

Output:



```
Coin Change Solution: ({25: 2, 10: 1, 1: 3}, 6)
Minimum File Merge Cost: 58
```

Priyanshu Sharma
CS24042

Practical 9

AIM: Divide and Conquer: Implement algorithms like merge sort and Strassen's Matrix Multiplication.

Definition:

Divide and Conquer

Divide and Conquer is an algorithmic technique used to solve problems by breaking them down into smaller subproblems. These subproblems are solved independently and their solutions are then combined to solve the original problem.

Merge Sort

Merge Sort is a **divide-and-conquer** algorithm that sorts an array or list by dividing it into two halves, sorting each half recursively, and then merging the sorted halves back together. It is a highly efficient, stable sorting algorithm with a time complexity of O(nlog⁡n)O(n \log n)O(nlogn).

Strassen's Matrix Multiplication

Strassen's Matrix Multiplication is an algorithm that multiplies two matrices more efficiently than the standard O(n3)O(n^3)O(n3) algorithm.

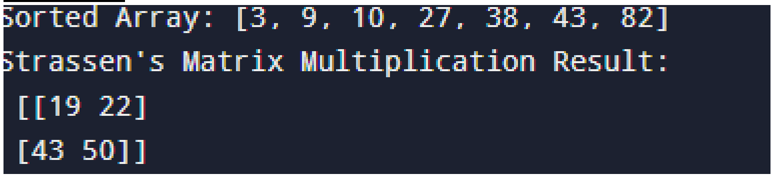
Code:

```
import numpy as np

# Merge Sort Algorithm (Divide and Conquer)
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)

def merge(left, right):
    sorted_arr = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1

    sorted_arr.extend(left[i:])
```

Priyanshu Sharma
CS24042

```
    sorted_arr.extend(right[j:])

    return sorted_arr

# Strassen's Matrix Multiplication (Divide and Conquer)
def strassen_multiply(A, B):
    assert A.shape == B.shape, "Matrices must be square and of same dimensions"
    n = A.shape[0]

    if n == 1:
        return A * B

    mid = n // 2

    A11, A12, A21, A22 = A[:mid, :mid], A[:mid, mid:], A[mid:, :mid], A[mid:, mid:]
    B11, B12, B21, B22 = B[:mid, :mid], B[:mid, mid:], B[mid:, :mid], B[mid:, mid:]

    M1 = strassen_multiply(A11 + A22, B11 + B22)
    M2 = strassen_multiply(A21 + A22, B11)
    M3 = strassen_multiply(A11, B12 - B22)
    M4 = strassen_multiply(A22, B21 - B11)
    M5 = strassen_multiply(A11 + A12, B22)
    M6 = strassen_multiply(A21 - A11, B11 + B12)
    M7 = strassen_multiply(A12 - A22, B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))
    return C

# Example Usage
if __name__ == "__main__":
    # Merge Sort Example
    arr = [38, 27, 43, 3, 9, 82, 10]
    print("Sorted Array:", merge_sort(arr))

    # Strassen's Matrix Multiplication Example
    A = np.array([[1, 2], [3, 4]])
    B = np.array([[5, 6], [7, 8]])
    print("Strassen's Matrix Multiplication Result:\n", strassen_multiply(A, B))
```

Output:



```
orted Array: [3, 9, 10, 27, 38, 43, 82]
trassen's Matrix Multiplication Result:
 [[19 22]
 [43 50]]
```

Priyanshu Sharma
CS24042

PRACTICAL 10

AIM: Dynamic Programming: Implement algorithms for Fibonacci series and Longest Common Subsequence using dynamic programming.

Definition: Dynamic Programming (DP) is an optimization technique used to solve complex problems by breaking them down into smaller subproblems, solving each subproblem only once, and storing their results to avoid redundant computations.

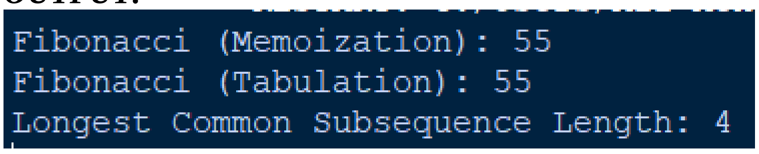
Code:

```
# Fibonacci Series using Dynamic Programming
# Memoization (Top-Down)
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]

# Tabulation (Bottom-Up)
def fibonacci_tab(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

# Longest Common Subsequence (LCS) using Dynamic Programming
def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example Usage
if __name__ == "__main__":
    # Fibonacci Example
    n = 10
    print("Fibonacci (Memoization):", fibonacci_memo(n))
    print("Fibonacci (Tabulation):", fibonacci_tab(n))

    # LCS Example
```

Priyanshu Sharma
CS24042

```
    str1 = "AGGTAB"
    str2 = "GXTXAYB"
    print("Longest Common Subsequence Length:", lcs(str1, str2))
```

OUTPUT:



```
Fibonacci (Memoization): 55
Fibonacci (Tabulation): 55
Longest Common Subsequence Length: 4
```