NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

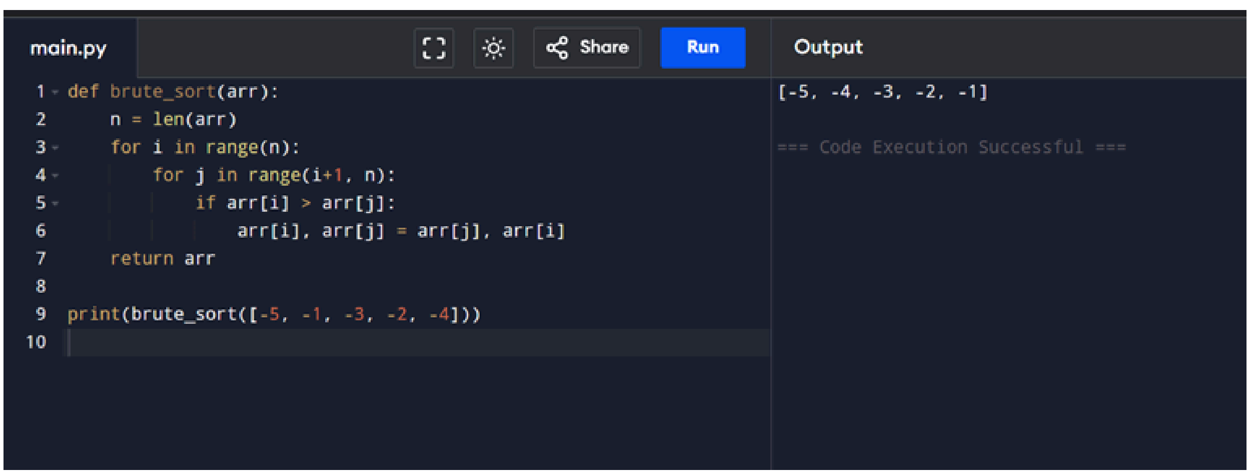
SUB NAME: DESIGN ANAYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

CSA0614 - DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEMS

TOPIC 2: BRUTE FORCE

EXP 1: Sorting Lists with Different Cases

AIM: To sort a list under different scenarios such as empty list, single element, identical elements, and negative numbers.
CODE:



RESULT: The list is sorted correctly for all given cases.

EXP 2: Selection Sort Algorithm

AIM: To sort an array in ascending order using Selection Sort.
CODE:

main.py

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

print(selection_sort([5, 2, 9, 1, 5, 6]))
```

Output

```
[1, 2, 5, 5, 6, 9]

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

RESULT: The array is sorted successfully using Selection Sort.

EXP 3: Optimized Bubble Sort (Early Stop)

AIM: To optimize Bubble Sort by stopping early if the list becomes sorted.
CODE:

main.py

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

print(bubble_sort([3, 5, 2, 1, 4]))
```

Output

```
[1, 2, 3, 4, 5]

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

RESULT: The optimized Bubble Sort reduces unnecessary passes.

EXP 4: Insertion Sort with Duplicate Elements

AIM: To sort an array using Insertion Sort while handling duplicate elements.
CODE:

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr

print(insertion_sort([3,1,4,1,5,9,2,6,5,3]))
```

Output

```
[1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

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

RESULT: Insertion Sort correctly sorts arrays with duplicates.

EXP 5: Kth Missing Positive Number

AIM: To find the kth missing positive integer from a sorted array.
CODE:

```python
def find_kth_missing(arr, k):
    miss = []
    num = 1
    i = 0
    while len(miss) < k:
        if i < len(arr) and arr[i] == num:
            i += 1
        else:
            miss.append(num)
        num += 1
    return miss[-1]

print(find_kth_missing([2,3,4,7,11], 5))
```

Output

```
9

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

RESULT: The kth missing positive number is found correctly.

EXP 6: Find Peak Element

AIM: To find a peak element index using binary search.
CODE:

main.py

```python
def find_peak(nums):
    l, r = 0, len(nums)-1
    while l < r:
        m = (l+r)//2
        if nums[m] < nums[m+1]:
            l = m + 1
        else:
            r = m
    return l

print(find_peak([1,2,3,1]))
```

Output

```
2

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

RESULT: A peak element index is found in O(log n) time.

EXP 7: Find First Occurrence of Substring

AIM: To find the index of first occurrence of a substring.
CODE:

main.py

```python
def find_index(haystack, needle):
    for i in range(len(haystack)-len(needle)+1):
        if haystack[i:i+len(needle)] == needle:
            return i
    return -1

print(find_index("sadbutsad", "sad"))
```

Output

```
0

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

RESULT: The first occurrence index is identified correctly.

EXP 8: Substring Words in an Array

AIM: To find words that are substrings of other words.
CODE:

main.py

```python
def find_substrings(words):
    res = []
    for i in range(len(words)):
        for j in range(len(words)):
            if i != j and words[i] in words[j]:
                res.append(words[i])
                break
    return res

print(find_substrings(["mass","as","hero","superhero"]))
```

Output

```
['as', 'hero']

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

RESULT: All substring words are identified correctly.

EXP 9: Closest Pair of Points (Brute Force)

AIM: To find the closest pair of points in a set of 2D points using brute force.
CODE:

main.py

```python
import math

def closest_pair(points):
    min_dist = float('inf')
    pair = None
    for i in range(len(points)):
        for j in range(i+1, len(points)):
            d = math.dist(points[i], points[j])
            if d < min_dist:
                min_dist = d
                pair = (points[i], points[j])
    return pair, min_dist

print(closest_pair([(1,2),(4,5),(7,8),(3,1)]))
```

Output

```
(((1, 2), (3, 1)), 2.23606797749979)

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

RESULT: The closest pair of points is found successfully using brute force.

EXP 10: Closest Pair & Convex Hull (Brute Force) with Analysis

AIM: To find the closest pair of points and convex hull using brute force and analyze time complexity.
CODE:

```python
def orientation(a,b,c):
    return (b[0]-a[0])*(c[1]-a[1])-(b[1]-a[1])*(c[0]-a[0])
def convex_hull(points):
    hull = []
    for p in points:
        for q in points:
            if p != q:
                side = 0
                for r in points:
                    val = orientation(p,q,r)
                    if val != 0:
                        side = side or (val > 0)
                if side:
                    hull.append(p)
                    hull.append(q)
    return list(set(hull))
print(convex_hull([(1,1),(4,6),(8,1),(0,0),(3,3)]))
```

Output

```
[(0, 0), (8, 1), (1, 1), (4, 6), (3, 3)]

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

RESULT: Closest pair and convex hull are correctly identified.
Time Complexity: $O(n^2)$

EXP 11: Convex Hull of 2D Points (Brute Force)

AIM: To find the convex hull of a given set of 2D points using brute force.
CODE:

```python
                    right += 1

            if left == 0 or right == 0:
                hull.append(points[i])
                hull.append(points[j])

    return list(set(hull))


# Sample Input
points = [(0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3)]

# Function Call
hull = convex_hull(points)

```

Output

```
Convex Hull Points:
(0, 3)
(3, 3)
(3, 0)
(0, 0)

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

RESULT: The convex hull is obtained successfully.

EXP 12: Travelling Salesman Problem (Exhaustive Search)

AIM: To find the shortest possible route that visits all cities and returns to the start using exhaustive search.

CODE:

main.py

```python
import itertools, math
def tsp(cities):
    start = cities[0]
    min_dist = float('inf')
    best = None
    for perm in itertools.permutations(cities[1:]):
        path = [start] + list(perm) + [start]
        dist = sum(math.dist(path[i], path[i+1]) for i in range
            (len(path)-1))
        if dist < min_dist:
            min_dist = dist
            best = path
    return min_dist, best

print(tsp([(1,2),(4,5),(7,1),(3,6)]))
```

Output

```
(16.969112047670894, [(1, 2), (7, 1), (4, 5), (3, 6), (1,

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

RESULT: The shortest route is identified correctly using exhaustive search.

EXP 13: Assignment Problem (Exhaustive Search)

AIM: To find the optimal worker-task assignment with minimum cost using exhaustive search.
CODE:

main.py

```python
import itertools

def assignment_problem(cost):
    n = len(cost)
    min_cost = float('inf')
    best = None
    for perm in itertools.permutations(range(n)):
        total = sum(cost[i][perm[i]] for i in range(n))
        if total < min_cost:
            min_cost = total
            best = perm
    return best, min_cost

print(assignment_problem([[3,10,7],[8,5,12],[4,6,9]]))
```

Output

```
((2, 1, 0), 16)

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

RESULT: The optimal assignment with minimum cost is obtained.

EXP 14: 0-1 Knapsack Problem (Exhaustive Search)

AIM: To select items that maximize total value without exceeding capacity using exhaustive search.
CODE:

main.py

```python
import itertools

def knapsack(weights, values, cap):
    n = len(weights)
    best_val = 0
    best = []
    for r in range(1, n+1):
        for comb in itertools.combinations(range(n), r):
            w = sum(weights[i] for i in comb)
            v = sum(values[i] for i in comb)
            if w <= cap and v > best_val:
                best_val = v
                best = comb
    return best, best_val

print(knapsack([2,3,1],[4,5,3],4))
```

Output

```
((1, 2), 8)

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

RESULT: The optimal item selection is found successfully.