
NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

SUB NAME: DESIGN ANAYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

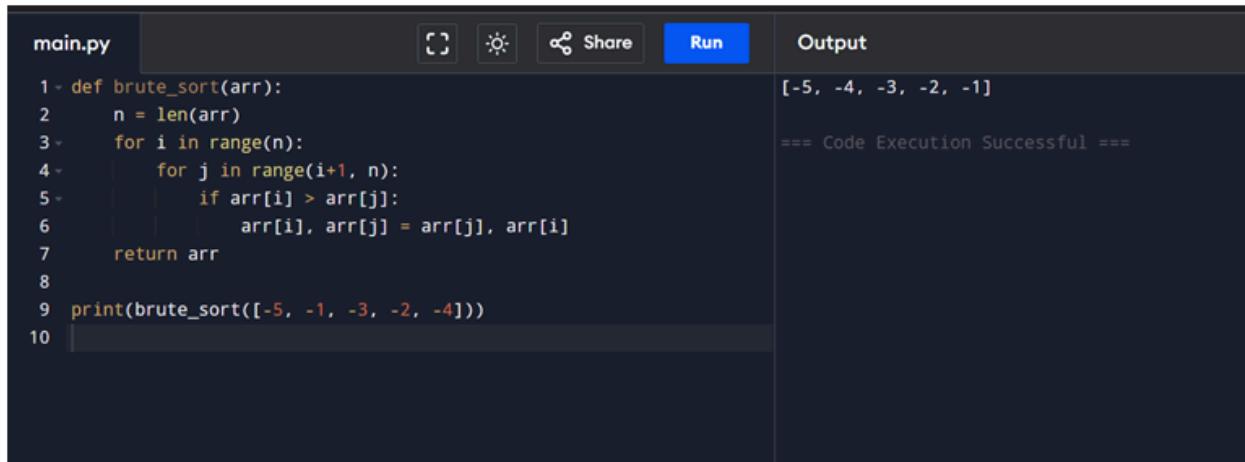
CSA0614 - DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEMS

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:



The screenshot shows a code editor interface with a dark theme. The left pane is titled "main.py" and contains the following Python code:

```
1 def brute_sort(arr):
2     n = len(arr)
3     for i in range(n):
4         for j in range(i+1, n):
5             if arr[i] > arr[j]:
6                 arr[i], arr[j] = arr[j], arr[i]
7     return arr
8
9 print(brute_sort([-5, -1, -3, -2, -4]))
10
```

The right pane is titled "Output" and displays the results of running the code. It shows the sorted array [-5, -4, -3, -2, -1] and a success message: "==== Code Execution Successful ===".

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:

The screenshot shows a Jupyter Notebook cell with the following content:

```
main.py
1 def selection_sort(arr):
2     n = len(arr)
3     for i in range(n):
4         min_idx = i
5         for j in range(i+1, n):
6             if arr[j] < arr[min_idx]:
7                 min_idx = j
8         arr[i], arr[min_idx] = arr[min_idx], arr[i]
9     return arr
10
11 print(selection_sort([5, 2, 9, 1, 5, 6]))
12
```

The "Run" button is highlighted. The "Output" section shows the sorted array [1, 2, 5, 5, 6, 9] and the message "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:

The screenshot shows a Jupyter Notebook cell with the following content:

```
main.py
1 def bubble_sort(arr):
2     n = len(arr)
3     for i in range(n):
4         swapped = False
5         for j in range(n-i-1):
6             if arr[j] > arr[j+1]:
7                 arr[j], arr[j+1] = arr[j+1], arr[j]
8                 swapped = True
9         if not swapped:
10             break
11     return arr
12
13 print(bubble_sort([3, 5, 2, 1, 4]))
14
```

The "Run" button is highlighted. The "Output" section shows the sorted array [1, 2, 3, 4, 5] and the message "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:

The screenshot shows a code editor window titled "main.py". The code implements the insertion sort algorithm. The output panel shows the sorted array [1, 1, 2, 3, 3, 4, 5, 5, 6, 9] and a success message.

```
1 def insertion_sort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i - 1
5         while j >= 0 and arr[j] > key:
6             arr[j+1] = arr[j]
7             j -= 1
8         arr[j+1] = key
9     return arr
10
11 print(insertion_sort([3,1,4,1,5,9,2,6,5,3]))
```

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:

The screenshot shows a code editor window titled "main.py". The code finds the kth missing positive integer in a sorted array. The output panel shows the result as 9 and a success message.

```
1 def find_kth_missing(arr, k):
2     miss = []
3     num = 1
4     i = 0
5     while len(miss) < k:
6         if i < len(arr) and arr[i] == num:
7             i += 1
8         else:
9             miss.append(num)
10        num += 1
11    return miss[-1]
12
13 print(find_kth_missing([2,3,4,7,11], 5))
```

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:

The screenshot shows a Jupyter Notebook interface with a dark theme. On the left, the code file 'main.py' contains a binary search algorithm to find a peak element in an array. The code is as follows:

```
1 def find_peak(nums):
2     l, r = 0, len(nums)-1
3     while l < r:
4         m = (l+r)//2
5         if nums[m] < nums[m+1]:
6             l = m + 1
7         else:
8             r = m
9     return l
10
11 print(find_peak([1,2,3,1]))
12
```

On the right, the 'Output' section shows the result of running the code: '2' and the message '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:

The screenshot shows a Jupyter Notebook interface with a dark theme. On the left, the code file 'main.py' contains a function 'find_index' to find the first occurrence of a substring 'needle' in a string 'haystack'. The code is as follows:

```
- 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"))
```

On the right, the 'Output' section shows the result of running the code: '0' and the message '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
```

```
1 def find_substrings(words):
2     res = []
3     for i in range(len(words)):
4         for j in range(len(words)):
5             if i != j and words[i] in words[j]:
6                 res.append(words[i])
7                 break
8     return res
9
10 print(find_substrings(["mass","as","hero","superhero"]))
11
```

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
```

```
1 import math
2
3 def closest_pair(points):
4     min_dist = float('inf')
5     pair = None
6     for i in range(len(points)):
7         for j in range(i+1, len(points)):
8             d = math.dist(points[i], points[j])
9             if d < min_dist:
10                 min_dist = d
11                 pair = (points[i], points[j])
12     return pair, min_dist
13
14 print(closest_pair([(1,2),(4,5),(7,8),(3,1)]))
15
```

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:

```

main.py
1 def orientation(a,b,c):
2     return (b[0]-a[0])*(c[1]-a[1])-(b[1]-a[1])*(c[0]-a[0])
3 def convex_hull(points):
4     hull = []
5     for p in points:
6         for q in points:
7             if p != q:
8                 side = 0
9                 for r in points:
10                     val = orientation(p,q,r)
11                     if val != 0:
12                         side = side or (val > 0)
13                 if side:
14                     hull.append(p)
15                     hull.append(q)
16     return list(set(hull))
17 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:

```

main.py
20
21
22     if left == 0 or right == 0:
23         hull.append(points[i])
24         hull.append(points[j])
25
26     return list(set(hull))
27
28
29 # Sample Input
30 points = [(0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3)]
31
32 # Function Call
33 hull = convex_hull(points)
34

```

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
```

```
1 import itertools, math
2 def tsp(cities):
3     start = cities[0]
4     min_dist = float('inf')
5     best = None
6     for perm in itertools.permutations(cities[1:]):
7         path = [start] + list(perm) + [start]
8         dist = sum(math.dist(path[i], path[i+1]) for i in range
9                    (len(path)-1))
9     if dist < min_dist:
10        min_dist = dist
11        best = path
12    return min_dist, best
13
14 print(tsp([(1,2),(4,5),(7,1),(3,6)]))
15
```

Output

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

```
1 import itertools
2
3 def assignment_problem(cost):
4     n = len(cost)
5     min_cost = float('inf')
6     best = None
7     for perm in itertools.permutations(range(n)):
8         total = sum(cost[i][perm[i]] for i in range(n))
9     if total < min_cost:
10        min_cost = total
11        best = perm
12    return best, min_cost
13
14 print(assignment_problem([[3,10,7],[8,5,12],[4,6,9]]))
15
```

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:

The screenshot shows a code editor interface with a dark theme. The left pane contains the Python file `main.py` with the following code:

```
1 import itertools
2
3 def knapsack(weights, values, cap):
4     n = len(weights)
5     best_val = 0
6     best = []
7     for r in range(1, n+1):
8         for comb in itertools.combinations(range(n), r):
9             w = sum(weights[i] for i in comb)
10            v = sum(values[i] for i in comb)
11            if w <= cap and v > best_val:
12                best_val = v
13                best = comb
14    return best, best_val
15
16 print(knapsack([2,3,1],[4,5,3],4))
17
```

The right pane shows the output of running the code. It displays the result of the `print` statement: `((1, 2), 8)`. Below this, a message indicates the execution was successful: `==== Code Execution Successful ===`.

RESULT: The optimal item selection is found successfully.