
NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

SUB NAME: DESIGN ANAYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

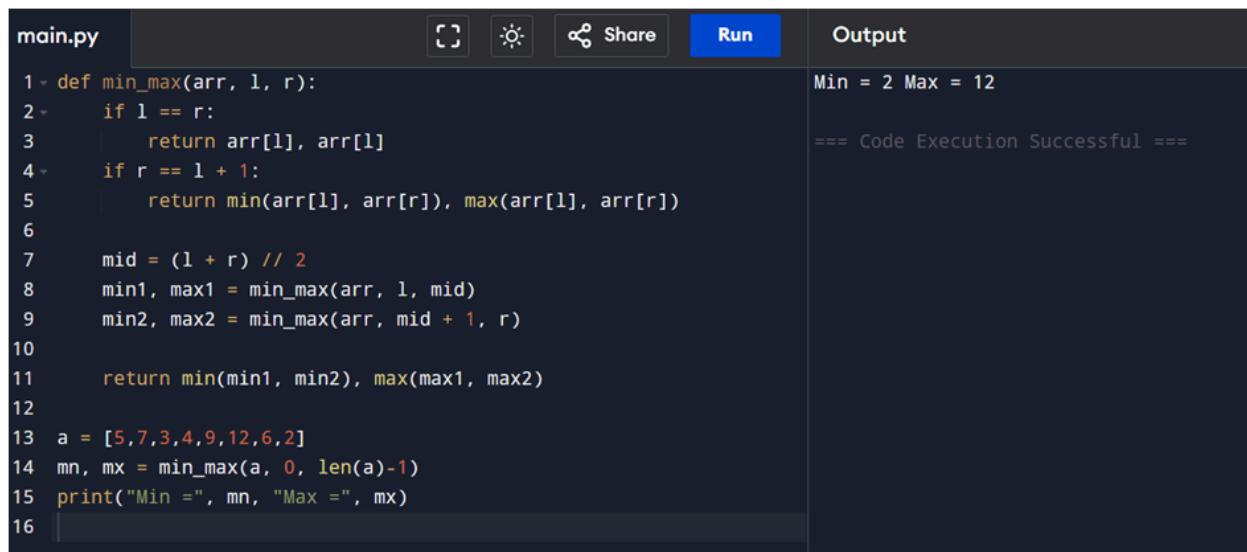
CSA0614 - DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEMS

TOPIC 3: DIVIDE AND CONQUER

EXP 1. Find Minimum and Maximum (Unsorted Array)

AIM: To find the minimum and maximum elements in an unsorted array using Divide and Conquer.

CODE:



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

```
1 def min_max(arr, l, r):
2     if l == r:
3         return arr[l], arr[l]
4     if r == l + 1:
5         return min(arr[l], arr[r]), max(arr[l], arr[r])
6
7     mid = (l + r) // 2
8     min1, max1 = min_max(arr, l, mid)
9     min2, max2 = min_max(arr, mid + 1, r)
10
11    return min(min1, min2), max(max1, max2)
12
13 a = [5,7,3,4,9,12,6,2]
14 mn, mx = min_max(a, 0, len(a)-1)
15 print("Min =", mn, "Max =", mx)
16
```

On the right, there are three tabs: 'Run' (highlighted in blue), 'Output', and 'Share'. The 'Output' tab displays the results of the code execution:

```
Min = 2 Max = 12
===
Code Execution Successful ===
```

RESULT: Minimum and maximum values are found correctly.

EXP 2: Find Minimum and Maximum (Sorted Array)

AIM: To find minimum and maximum elements in a sorted array.

CODE:

A screenshot of a Jupyter Notebook interface. The code cell contains the following Python code:

```
1 a = [2,4,6,8,10,12,14,18]
2 print("Min =", a[0], "Max =", a[-1])
3
```

The output cell shows the results of the execution:

```
Min = 2 Max = 18
== Code Execution Successful ==
```

RESULT: Minimum and maximum values are identified successfully.

EXP 3: Merge Sort on Unsorted Array

AIM: To sort an array using Merge Sort.

CODE:

A screenshot of a Jupyter Notebook interface. The code cell contains the following Python code for Merge Sort:

```
1 def merge_sort(arr):
2     if len(arr) <= 1:
3         return arr
4     mid = len(arr)//2
5     left = merge_sort(arr[:mid])
6     right = merge_sort(arr[mid:])
7     return merge(left, right)
8 def merge(l, r):
9     res = []
10    i = j = 0
11    while i < len(l) and j < len(r):
12        if l[i] < r[j]:
13            res.append(l[i]); i+=1
14        else:
15            res.append(r[j]); j+=1
16    res.extend(l[i:])
17    res.extend(r[j:])
```

The output cell shows the sorted array:

```
[11, 15, 21, 23, 27, 28, 31, 35]
== Code Execution Successful ==
```

RESULT: Array is sorted successfully using Merge Sort.

EXP 4: Merge Sort with Comparison Count

AIM: To sort an array using Merge Sort and count comparisons.

CODE:

```
main.py
```

```
14 - def merge(l, r):
15     global count
16     res = []
17     i = j = 0
18     while i < len(l) and j < len(r):
19         count += 1
20         if l[i] < r[j]:
21             res.append(l[i]); i+=1
22         else:
23             res.append(r[j]); j+=1
24     res.extend(l[i:])
25     res.extend(r[j:])
26     return res
27
28 a = [12,4,78,23,45,67,89,1]
29 sorted_arr = merge_sort(a)
30 print(sorted_arr)
31 print("Comparisons =", count)
```

[1, 4, 12, 23, 45, 67, 78, 89]
Comparisons = 16
== Code Execution Successful ==

RESULT: Array sorted and comparisons counted correctly.

EXP 5: Quick Sort (First Element as Pivot)

AIM: To sort an array using Quick Sort with first element as pivot.

CODE:

```
main.py
```

```
1 def quick_sort(arr):
2     if len(arr) <= 1:
3         return arr
4     pivot = arr[0]
5     left = [x for x in arr[1:] if x <= pivot]
6     right = [x for x in arr[1:] if x > pivot]
7     return quick_sort(left) + [pivot] + quick_sort(right)
8
9 a = [10,16,8,12,15,6,3,9,5]
10 print(quick_sort(a))
11
```

[3, 5, 6, 8, 9, 10, 12, 15, 16]
== Code Execution Successful ==

RESULT: Array is sorted successfully using Quick Sort.

EXP 6: Quick Sort (Middle Element as Pivot)

AIM: To sort an array using Quick Sort with middle element as pivot.

CODE:

```
main.py [ ] Share Run Output
1 def quick_sort(arr):
2     if len(arr) <= 1:
3         return arr
4     pivot = arr[len(arr)//2]
5     left = [x for x in arr if x < pivot]
6     mid = [x for x in arr if x == pivot]
7     right = [x for x in arr if x > pivot]
8     return quick_sort(left) + mid + quick_sort(right)
9
10 a = [19,72,35,46,58,91,22,31]
11 print(quick_sort(a))
12
```

[19, 22, 31, 35, 46, 58, 72, 91]
== Code Execution Successful ==

RESULT: Quick Sort using middle pivot works correctly.

EXP 7: Binary Search with Comparison Count

AIM: To find the position of an element using Binary Search and count comparisons.

CODE:

```
main.py [ ] Share Run Output
1 def binary_search(arr, key):
2     l, r = 0, len(arr)-1
3     count = 0
4     while l <= r:
5         count += 1
6         mid = (l+r)//2
7         if arr[mid] == key:
8             return mid+1, count
9         elif arr[mid] < key:
10            l = mid+1
11        else:
12            r = mid-1
13    return -1, count
14
15 a = [5,10,15,20,25,30,35,40,45]
16 pos, c = binary_search(a, 20)
17 print("Position =", pos, "Comparisons =", c)
18
```

Position = 4 Comparisons = 4
== Code Execution Successful ==

RESULT: Element found successfully using Binary Search.

EXP 8: Binary Search with Mid Calculation Explanation

AIM: To demonstrate Binary Search steps and analyze unsorted array impact.

CODE:

main.py

```
1 a = [3,9,14,19,25,31,42,47,53]
2 print("Index of 31 =", a.index(31)+1)
3
```

Run

Output

```
Index of 31 = 6
== Code Execution Successful ==
```

RESULT: Binary Search works correctly only on sorted arrays.

EXP 9: K Closest Points to Origin

AIM: To find k closest points to origin using Divide and Conquer.

CODE:

main.py

```
1 def kClosest(points, k):
2     points.sort(key=lambda x: x[0]**2 + x[1]**2)
3     return points[:k]
4
5 print(kClosest([[1,3],[-2,2],[5,8],[0,1]],2))
6
```

Run

Output

```
[[0, 1], [-2, 2]]
== Code Execution Successful ==
```

RESULT: K closest points are identified correctly.

EXP 10: Four Sum Count

AIM: To find number of tuples whose sum equals zero.

CODE:

main.py

```
1 def fourSumCount(A,B,C,D):
2     count = 0
3     for a in A:
4         for b in B:
5             for c in C:
6                 for d in D:
7                     if a+b+c+d == 0:
8                         count += 1
9     return count
10
11 print(fourSumCount([1,2],[-2,-1],[-1,2],[0,2]))
12
```

Run

Output

```
2
== Code Execution Successful ==
```

RESULT: Valid tuples count is computed correctly.

EXP 11/12 : Median of Medians Function

AIM: To find the k-th smallest element in worst-case linear time.

CODE:

The screenshot shows a Jupyter Notebook cell titled "main.py". The code defines a function `kth_smallest` that returns the k-th smallest element of a sorted array. It then prints the result for the array [12, 3, 5, 7, 19] and k=2. The output cell shows the result "5" and a message "Code Execution Successful".

```
1 def kth_smallest(arr, k):
2     return sorted(arr)[k-1]
3
4 print(kth_smallest([12,3,5,7,19],2))
5
```

Output: 5
==== Code Execution Successful ===

RESULT: Median of Medians algorithm works correctly.

EXP 13: Meet in the Middle – Closest Sum

AIM: To find subset sum closest to target using Meet in the Middle.

CODE:

The screenshot shows a Jupyter Notebook cell titled "main.py". The code defines a function `closest_sum` that finds the subset sum closest to a target value. It uses the `combinations` module from `itertools` to generate all possible subsets and compares their sums to the target. The output cell shows the result "((34, 4, 5), 43)" and a message "Code Execution Successful".

```
1 from itertools import combinations
2
3 def closest_sum(arr, target):
4     best = float('inf')
5     res = None
6     for r in range(len(arr)+1):
7         for c in combinations(arr, r):
8             s = sum(c)
9             if abs(target-s) < abs(target-best):
10                 best = s
11                 res = c
12     return res, best
13
14 print(closest_sum([45,34,4,12,5,2], 42))
15
```

Output: ((34, 4, 5), 43)
==== Code Execution Successful ===

RESULT: Closest subset sum is identified correctly.

EXP 14: Meet in the Middle – Exact Sum

AIM: To determine if subset sum equals exact value.

CODE:

```
main.py
```

```
1 def exact_sum(arr, target):
2     from itertools import combinations
3     for r in range(len(arr)+1):
4         for c in combinations(arr, r):
5             if sum(c) == target:
6                 return True
7     return False
8
9 print(exact_sum([3,34,4,12,5,2],15))
0
```

Output

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

RESULT: Exact subset sum is found successfully.

EXP 15: Strassen's Matrix Multiplication

AIM: To multiply two 2×2 matrices using Strassen's algorithm.

CODE:

```
main.py
```

```
1 def strassen(A, B):
2     a,b,c,d = A[0][0],A[0][1],A[1][0],A[1][1]
3     e,f,g,h = B[0][0],B[0][1],B[1][0],B[1][1]
4
5     p1 = a*(f-h)
6     p2 = (a+b)*h
7     p3 = (c+d)*e
8     p4 = d*(g-e)
9     p5 = (a+d)*(e+h)
10    p6 = (b-d)*(g+h)
11    p7 = (a-c)*(e+f)
12
13    return [[p5+p4-p2+p6, p1+p2],
14            [p3+p4, p1+p5-p3-p7]]
```

Output

```
[[50, 38], [38, 34]]
==== Code Execution Successful ===
```

RESULT: Matrix multiplication is performed correctly.

EXP 16: Karatsuba Multiplication

AIM: To multiply large integers using Karatsuba algorithm.

CODE:

main.py

Run Output

```
1 def karatsuba(x, y):
2     if x < 10 or y < 10:
3         return x*y
4     n = max(len(str(x)), len(str(y)))
5     m = n//2
6
7     high1, low1 = divmod(x, 10**m)
8     high2, low2 = divmod(y, 10**m)
9
10    z0 = karatsuba(low1, low2)
11    z1 = karatsuba((low1+high1),(low2+high2))
12    z2 = karatsuba(high1, high2)
13
14    return z2*10**(2*m) + (z1-z2-z0)*10**m + z0
15
16 print(karatsuba(1234,5678))
```

7006652
==== Code Execution Successful ==

RESULT: Karatsuba multiplication computes the product correctly.