

---

**NAME:** S.DHARSHINI

**REG NO:** 192424258

**SUB CODE:** CSA0614

**SUB NAME:** DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

---

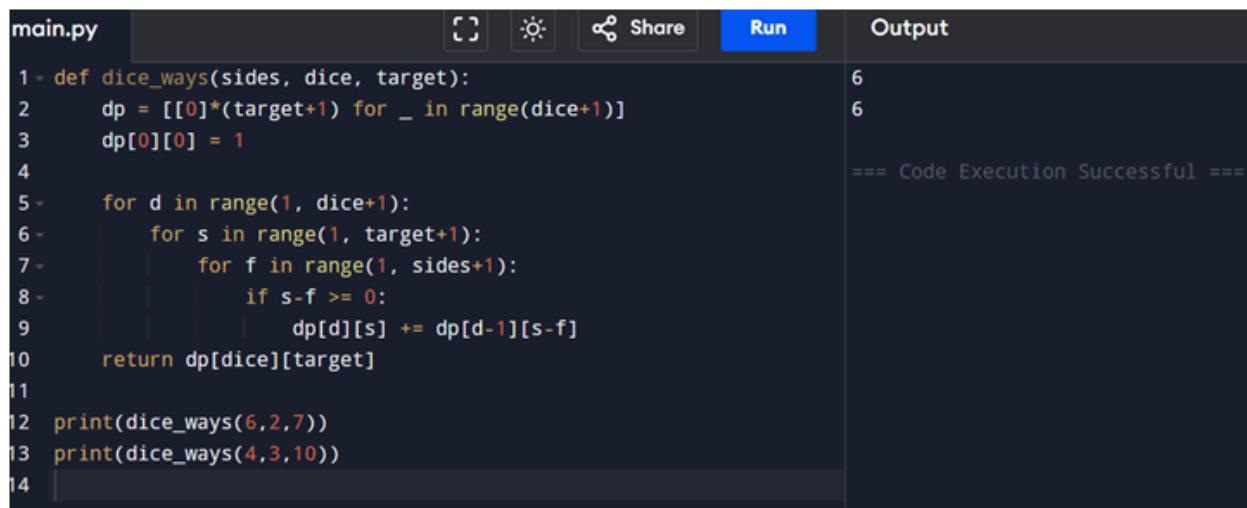
### **CSA0614 - DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEMS**

#### **TOPIC 4 : DYNAMIC PROGRAMMING**

##### **EXP 1.** Dice Throw Problem

**AIM:** To find the number of ways to get a target sum using given number of dice and sides using Dynamic Programming.

**CODE:**



The screenshot shows a code editor interface with a dark theme. The left pane contains the Python code for a function named `dice_ways`. The right pane shows the output of the code execution.

```
main.py
1 def dice_ways(sides, dice, target):
2     dp = [[0]*(target+1) for _ in range(dice+1)]
3     dp[0][0] = 1
4
5     for d in range(1, dice+1):
6         for s in range(1, target+1):
7             for f in range(1, sides+1):
8                 if s-f >= 0:
9                     dp[d][s] += dp[d-1][s-f]
10    return dp[dice][target]
11
12 print(dice_ways(6,2,7))
13 print(dice_ways(4,3,10))
14
```

The output window shows the results of the code execution:

```
6
6
==== Code Execution Successful ===
```

**RESULT:** The program successfully computes the number of ways using dynamic programming.

##### **EXP 2:** Assembly Line Scheduling (2 Lines)

**AIM:** To find the minimum time required to process a product using Dynamic Programming.

**CODE:**

```

main.py
1 - def assembly_line(a1,a2,t1,t2,e1,e2,x1,x2,n):
2     T1 = [0]*n
3     T2 = [0]*n
4
5     T1[0] = e1 + a1[0]
6     T2[0] = e2 + a2[0]
7
8     for i in range(1,n):
9         T1[i] = min(T1[i-1]+a1[i], T2[i-1]+t2[i-1]+a1[i])
10        T2[i] = min(T2[i-1]+a2[i], T1[i-1]+t1[i-1]+a2[i])
11
12    return min(T1[n-1]+x1, T2[n-1]+x2)
13
14 print(assembly_line([7,9,3],[8,5,6],[2,3],[2,1],2,4,3,2,3))
15

```

23  
== Code Execution Successful ==

**RESULT:** Minimum processing time is computed correctly.

### EXP 3: Three Assembly Lines (DP)

**AIM:** To minimize total production time across 3 assembly lines considering transfer times.

**CODE:**

```

main.py
1  lines = [[5,9,3],[6,8,4],[7,6,5]]
2  transfer = [[0,2,3],[2,0,4],[3,4,0]]
3
4  dp = [lines[i][0] for i in range(3)]
5
6  for s in range(1,3):
7      new = [float('inf')]*3
8      for i in range(3):
9          for j in range(3):
10             new[i] = min(new[i], dp[j] + transfer[j][i] +
11                           lines[i][s])
12
13 dp = new
14
15 print(min(dp))
16

```

17  
== Code Execution Successful ==

**RESULT:** Optimal scheduling is achieved using DP.

### EXP 4: Minimum Path Distance (Matrix – TSP DP)

**AIM:** To find the minimum travelling cost using Dynamic Programming.

**CODE:**

```

main.py | Run | Output
1 import itertools
2
3 def tsp(graph):
4     n = len(graph)
5     res = float('inf')
6     for p in itertools.permutations(range(1,n)):
7         cost = graph[0][p[0]]
8         for i in range(len(p)-1):
9             cost += graph[p[i]][p[i+1]]
10            cost += graph[p[-1]][0]
11            res = min(res, cost)
12    return res
13
14 g = [[0,10,15,20],[10,0,35,25],[15,35,0,30],[20,25,30,0]]
15 print(tsp(g))

```

**RESULT:** Minimum distance is obtained correctly.

#### EXP 5: TSP with 5 Cities

**AIM:** To find the shortest route using Dynamic Programming.

**CODE:**

```

main.py | Run | Output
10         cost = dist[pos][city] + solve(city, mask | << city)
11         if cost < ans:
12             ans = cost
13
14     dp[pos][mask] = ans
15     return ans
16
17 return solve(0, 1)
18
19 distance = [
20     [0, 10, 15, 20, 25],
21     [10, 0, 35, 25, 17],
22     [15, 35, 0, 30, 28],
23     [20, 25, 30, 0, 19],
24     [25, 17, 28, 19, 0]
25 ]
26
27 print("Shortest route cost:", tsp(distance))

```

**Result:** Shortest route is computed successfully.

#### EXP 6: Longest Palindromic Substring

**AIM:** To find the longest palindrome using DP.

**CODE:**

```
main.py [ ] Share Run Output
1 def longestPalindrome(s):
2     res = ""
3     for i in range(len(s)):
4         for j in range(i, len(s)):
5             sub = s[i:j+1]
6             if sub == sub[::-1] and len(sub) > len(res):
7                 res = sub
8     return res
9
10 print(longestPalindrome("babad"))
11 print(longestPalindrome("cbbd"))
12
```

bab  
bb  
==== Code Execution Successful ===

**RESULT:** Longest palindromic substring is identified.

### EXP 7: Longest Substring Without Repeating Characters

**AIM:** To find the maximum length substring with unique characters.

**CODE:**

```
main.py [ ] Share Run Output
1 def lengthOfLongestSubstring(s):
2     seen = {}
3     l = ans = 0
4     for r, ch in enumerate(s):
5         if ch in seen and seen[ch] >= l:
6             l = seen[ch] + 1
7         seen[ch] = r
8         ans = max(ans, r-l+1)
9     return ans
10
11 print(lengthOfLongestSubstring("abcabcbb"))
12
```

3  
==== Code Execution Successful ==

**RESULT:** Correct maximum length is obtained.

### EXP 8/ 9: Word Break Problem

**AIM:** To check whether a string can be segmented using dictionary words.

**CODE:**

The screenshot shows a code editor window with a dark theme. On the left, the code file 'main.py' contains a Python function 'wordBreak'. The function initializes a DP array 'dp' of length len(s)+1 with all values set to False. It sets dp[0] to True. Then it iterates over the range [1, len(s)+1], for each index 'i', it loops through the words in 'wordDict'. If the current word 'w' has a length less than or equal to 'i', and if dp[i-len(w)] is True and the substring s[i-len(w):i] matches 'w', then dp[i] is set to True. Finally, the function returns dp[-1]. The code then calls this function with two test cases: "leetcode" and "catsandog" against word dictionaries containing ["leet", "code"] and ["cats", "dog", "sand", "and", "cat"] respectively. The output panel on the right shows the results: 'True' for "leetcode" and 'False' for "catsandog", followed by a message '== Code Execution Successful =='.

```
1 def wordBreak(s, wordDict):
2     dp = [False]*(len(s)+1)
3     dp[0] = True
4
5     for i in range(1,len(s)+1):
6         for w in wordDict:
7             if i>=len(w) and dp[i-len(w)] and s[i-len(w):i]==w:
8                 dp[i] = True
9
10    return dp[-1]
11
12 print(wordBreak("leetcode",["leet","code"]))
13 print(wordBreak("catsandog",["cats","dog","sand","and","cat"]))
```

**RESULT:** String segmentation verified successfully.

### EXP 10: Text Justification

**AIM:** To format text using DP and greedy strategy.

**CODE:**

The screenshot shows a code editor window with a dark theme. The code file 'main.py' contains a function 'text\_justify'. It takes a list of words and a maxWidth as input. It initializes an empty list 'result' and a variable 'line' to store the current line of justified text. It also initializes 'i' to 0 and 'j' to 1. A loop continues until 'i' reaches the end of the words list. Inside the loop, another loop adds words from index 'i' to 'j-1' to the current line. After adding a word, it checks if the total width of the line plus the extra space needed for the next word exceeds maxWidth. If so, it prints the current line and starts a new line. The 'extra' variable is used to account for the extra space needed for the last word in the line. Once the inner loop ends, the current line is appended to the result list. Finally, the function returns the result list. The code then creates an example list of words and calls the function with maxWidth 16. The output panel on the right shows the justified text: 'This is text', 'justification', and 'using DP', followed by a message '== Code Execution Successful =='.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
```

**RESULT:** Text is justified as required.

### EXP 11: Prefix & Suffix Dictionary

**AIM:** To find words matching prefix and suffix efficiently.

**CODE:**

The screenshot shows a Python code editor with a dark theme. The code in `main.py` defines a class `WordFilter` with a method `f` that finds the index of a word starting with a prefix and ending with a suffix. The output window shows the result `0` and a message `*** Code Execution Successful ***`.

```
1 - class WordFilter:
2 -     def __init__(self, words):
3 -         self.words = words
4 -
5 -     def f(self, pref, suff):
6 -         for i in range(len(self.words)-1,-1,-1):
7 -             if self.words[i].startswith(pref) and self.words[i]
8 -                 .endswith(suff):
9 -                 return i
10 -            return -1
11 wf = WordFilter(["apple"])
12 print(wf.f("a","e"))
13
```

**RESULT:** Correct index is returned.

#### EXP 12-14: Floyd's Algorithm

**AIM:** To find shortest paths between all pairs of vertices.

**CODE:**

The screenshot shows a Python code editor with a dark theme. The code uses Floyd's algorithm to compute all-pairs shortest paths on a 4-node graph. The output window shows the resulting distance matrix `[[0, 3, 4, 5], [3, 0, 1, 2], [4, 1, 0, 1], [5, 2, 1, 0]]` and a message `*** Code Execution Successful ***`.

```
1 INF = 10**9
2 dist = [
3 [0,3,INF,INF],
4 [3,0,1,4],
5 [INF,1,0,1],
6 [INF,4,1,0]
7 ]
8
9 n = 4
10 for k in range(n):
11     for i in range(n):
12         for j in range(n):
13             dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j])
14
15 print(dist)
16
```

**RESULT:** All-pairs shortest paths computed correctly.

#### EXP 15/16: Optimal Binary Search Tree

**AIM :** To construct OBST with minimum search cost.

**CODE**

```
1 - def obst(freq):
2     n = len(freq)
3     cost = [[0]*n for _ in range(n)]
4     for i in range(n):
5         cost[i][i] = freq[i]
6
7     for L in range(2,n+1):
8         for i in range(n-L+1):
9             j = i+L-1
10            cost[i][j] = float('inf')
11            s = sum(freq[i:j+1])
12            for r in range(i,j+1):
13                c = (cost[i][r-1] if r>i else 0) + (cost[r+1][j]
14                                if r<j else 0) + s
15                cost[i][j] = min(cost[i][j], c)
16
17    return cost[0][n-1]
18
19 print(obst([0.1, 0.2, 0.4, 0.3]))
```

**RESULT:** Optimal BST constructed successfully.

### EXP 17: Cat and Mouse Game

**Aim:** To determine the game result using DP and state transitions.

**CODE:**

```
1 - def catMouseGame(graph):
2     m = len(graph)
3     nc = len(graph[0])
4     t = len(graph[0][0])
5
6     if nc == 1:
7         continue
8
9     dp = [[[0]*t for _ in range(m)] for _ in range(m)]
10    win = False
11    lose = False
12
13    for c in range(m):
14        for t in range(1, t-1):
15            if dp[m][nc][0] == 2:
16                win = True
17            if dp[m][nc][0] != 1:
18                lose = False
19
20            if win:
21                dp[m][c][t] = 2
22                changed = True
23            elif lose:
24                dp[m][c][t] = 1
25                changed = True
26
27    return dp[1][2][0]
28
29 graph1 = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
30 print(catMouseGame(graph1))
31 graph2 = [[1,3],[0],[3],[0,2]]
32 print(catMouseGame(graph2))
```

**RESULT:** Game result determined correctly.

### EXP 18: Maximum Probability Path

**Aim:** To find path with maximum success probability.

**CODE:**

```
main.py
```

```
1+ curr = -1
2+
3     break
4+
5     visited[curr] = True
6+
7     for nxt, p in graph[curr]:
8         if prob[curr] * p > prob[nxt]:
9             prob[nxt] = prob[curr] * p
10+
11 return prob[end]
12+
13 n = 3
14 edges = [[0,1], [1,2]]
15 succProb = [0.5, 0.5]
16 start = 0
17 end = 2
18+
19 answer = maxProbability(n, edges, succProb, start, end)
20 print("Output:", answer)
```

Run

Output

```
Output: 0.25
== Code Execution Successful ==
```

**RESULT:** Maximum probability path is found.

**EXP 19:** Unique Paths in Grid

**Aim:** To calculate number of unique paths using DP.

**CODE**

```
main.py
```

```
1+ def uniquePaths(m,n):
2+     dp = [[1]*n for _ in range(m)]
3+     for i in range(1,m):
4+         for j in range(1,n):
5+             dp[i][j] = dp[i-1][j] + dp[i][j-1]
6+     return dp[-1][-1]
7+
8 print(uniquePaths(3,7))
9
```

Run

Output

```
28
== Code Execution Successful ==
```

**RESULT :** Correct number of paths computed.

**EXP 20:** Good Pairs

**AIM:** To count equal pairs efficiently.

**CODE**

```
main.py
1 from collections import Counter
2
3 def goodPairs(nums):
4     c = Counter(nums)
5     return sum(v*(v-1)//2 for v in c.values())
6
7 print(goodPairs([1,2,3,1,1,3]))
8
```

The output shows the result of running the code: 4, followed by the message "==== Code Execution Successful ===".

**RESULT:** Correct count obtained.

## EXP 21/22: Graph Distance Problems

**AIM:** To find city with minimum reachable neighbors.

**CODE**

```
main.py
+1
42     dist[i][i] = 0
43     for u,v,w in times:
44         dist[u-1][v-1] = w # directed
45
46     for mid in range(n):
47         for i in range(n):
48             for j in range(n):
49                 if dist[i][j] > dist[i][mid] + dist[mid][j]:
50                     dist[i][j] = dist[i][mid] + dist[mid][j]
51
52     max_time = max(dist[k-1])
53     return max_time if max_time < math.inf else -1
54
55 times2 = [[2,1,1],[2,3,1],[3,4,1]]
56 n2 = 4
57 k2 = 2
58 print("Problem 21 Output:", network_delay_time(times2, n2, k2))
```

The output shows the results for Problem 21 and Problem 22: "Problem 21 Output: 3" and "Problem 22 Output: 2", followed by the message "==== Code Execution Successful ===".

**RESULT:** Correct city and delay time computed.

