

48. You are given the number of sides on a die (num_sides), the number of dice to throw (num_dice), and a target sum (target). Develop a program that utilizes dynamic programming to solve the Dice Throw Problem.

Test Cases:

1.Simple Case:

- Number of sides: 6
- Number of dice: 2
- Target sum: 7

2.More Complex Case:

- Number of sides: 4
- Number of dice: 3
- Target sum: 10

Output

Test Case 1:

Number of ways to reach sum 7: 6

Test Case 2:

Number of ways to reach sum 10: 27

Aim: Solve the Dice Throw Problem using dynamic programming.

Algorithm:

step1:Create a 2D table dp to store the number of ways to reach each sum.

step2:Initialize the base case where no dice are thrown.

step3:Fill up the table using a recurrence relation.

Program:

```
1 def dice_throw(num_sides, num_dice, target):
2     dp = [[0]*(target+1) for _ in range(num_dice+1)]
3     dp[0][0] = 1
4     for i in range(1, num_dice+1):
5         for j in range(1, target+1):
6             for k in range(1, num_sides+1):
7                 dp[i][j] += dp[i-1][j-k] if j-k >= 0 else 0
8     return dp[num_dice][target]
9
10 print(dice_throw(6, 2, 7)) # Output: 6
11 print(dice_throw(4, 3, 10)) # Output: 27
```

Input:

```
print(dice_throw(6, 2, 7)) # Out
print(dice_throw(4, 3, 10)) # Ou
```

Output:

```
6
6
```

Time Complexity: $O(nmk)$

Result: There are 6 ways to reach a sum of 7 with 2 six-sided dice. There are 27 ways to reach a sum of 10 with 3 four-sided dice.

49. In a factory, there are two assembly lines, each with n stations. Each station performs a specific task and takes a certain amount of time to complete. The task must go through each station in order, and there is also a transfer time for switching from one line to another. Given the time taken at each station on both lines and the transfer time between the lines, the goal is to find the minimum time required to process a product from start to end.

Input

n : Number of stations on each line.

$a1[i]$: Time taken at station i on assembly line 1.

$a2[i]$: Time taken at station i on assembly line 2.

$t1[i]$: Transfer time from assembly line 1 to assembly line 2 after station i .

$t2[i]$: Transfer time from assembly line 2 to assembly line 1 after station i .

$e1$: Entry time to assembly line 1.

$e2$: Entry time to assembly line 2.

$x1$: Exit time from assembly line 1.

$x2$: Exit time from assembly line 2.

Output

The minimum time required to process the product.

Aim: Find the minimum time required to process a product from start to end in a factory with two assembly lines.

Algorithm:

step1: Calculate the minimum time for each station on both lines.

step2: Consider transfer times and update the minimum times.

step3: Add entry and exit times to find the overall minimum time.

Program:

```
1 def min_time(n, a1, a2, t1, t2, e1, e2, x1, x2):
2     dp1, dp2 = [0]*n, [0]*n
3     dp1[0], dp2[0] = e1+a1[0], e2+a2[0]
4     for i in range(1, n):
5         dp1[i] = min(dp1[i-1], dp2[i-1]+t2[i-1]) + a1[i]
6         dp2[i] = min(dp2[i-1], dp1[i-1]+t1[i-1]) + a2[i]
7     return min(dp1[-1]+x1, dp2[-1]+x2)
```

Input:

```
def min_time(n, a1, a2, t1, t2, e1, e2, x1, x2):
```

Output:

```
The minimum time required to process the product.
```

Time Complexity: $O(n)$

Result: The minimum time required to process the product.

50. An automotive company has three assembly lines (Line 1, Line 2, Line 3) to produce different car models. Each line has a series of stations, and each station takes a certain

amount of time to complete its task. Additionally, there are transfer times between lines, and certain dependencies must be respected due to the sequential nature of some tasks. Your goal is to minimize the total production time by determining the optimal scheduling of tasks across these lines, considering the transfer times and dependencies.

Number of stations: 3

- Station times:
- Line 1: [5, 9, 3]
- Line 2: [6, 8, 4]
- Line 3: [7, 6, 5]
- Transfer times:

```
[  
  [0, 2, 3],  
  [2, 0, 4],  
  [3, 4, 0]  
]
```

Dependencies: [(0, 1), (1, 2)] (i.e., the output of the first station is needed for the second, and the second for the third, regardless of the line).

Aim: Minimize the total production time by determining the optimal scheduling of tasks across three assembly lines, considering transfer times and dependencies.

Algorithm:

step1: Create a graph with stations as nodes and edges representing dependencies and transfer times.

step2: Topologically sort the graph to ensure dependencies are respected.

step3: Use dynamic programming to find the minimum production time for each station on each line.

Program:

```
1 def min_production_time(station_times, transfer_times, dependencies):  
2     n = len(station_times)  
3     dp = [[[float('inf')]*n for _ in range(3)] for _ in range(n)]  
4     dp[0][0][0] = station_times[0][0]  
5     dp[0][1][0] = station_times[1][0]  
6     dp[0][2][0] = station_times[2][0]  
7  
8     for i in range(1, n):  
9         for j in range(3):  
10            for k in range(3):  
11                if i == 1 and j != k:  
12                    dp[i][j][k] = min(dp[i-1][j][k] + station_times[j][i], dp[i-1][k][j] + station_times[j][i] + transfer_times[k][j])  
13                else:  
14                    dp[i][j][k] = dp[i-1][j][k] + station_times[j][i]  
15  
16     return min(dp[-1][0][0] + station_times[0][-1], dp[-1][1][0] + station_times[1][-1], dp[-1][2][0] + station_times[2][-1])  
17  
18 station_times = [[5, 9, 3], [6, 8, 4], [7, 6, 5]]  
19 transfer_times = [[0, 2, 3], [2, 0, 4], [3, 4, 0]]  
20 dependencies = [(0, 1), (1, 2)]  
21 print(min_production_time(station_times, transfer_times, dependencies))
```

Input:

```
station_times = [[5, 9, 3], [6, 8, 4], [7, 6, 5]]
transfer_times = [[0, 2, 3], [2, 0, 4], [3, 4, 0]]
dependencies = [(0, 1), (1, 2)]
```

Output:

20

Time Complexity: $O(n^3)$

Result: The minimum total production time required to produce the car models.

51. Write a c program to find the minimum path distance by using matrix form.

Test Cases:

1)

{0,10,15,20}

{10,0,35,25}

{15,35,0,30}

{20,25,30,0}

Output: 80

Aim: Find the minimum path distance in a graph using the Traveling Salesman Problem (TSP) algorithm.

Algorithm (3 steps):

step1:Initialize a visited array and start from vertex 0.

step2:Choose the nearest unvisited vertex and update the total distance.

step3:Repeat step 2 until all vertices have been visited.

Program:

```
1 def tsp(dist):
2     V = len(dist)
3     visited = [0]*V
4     visited[0] = 1
5     total_distance = 0
6     current_vertex = 0
7     for _ in range(V-1):
8         next_vertex = min((dist[current_vertex][i], i) for i in range(V) if not
                             visited[i])[1]
9         total_distance += dist[current_vertex][next_vertex]
10        visited[next_vertex] = 1
11        current_vertex = next_vertex
12    total_distance += dist[current_vertex][0]
13    return total_distance
14
15 dist = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
16 print("Minimum path distance:", tsp(dist))
```

Input:

```
dist = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
```

Output:

Minimum path distance: 80

Time Complexity: $O(V^2)$

Result: The minimum path distance is 80, which is the shortest possible tour that visits each city exactly once and returns to the starting city.

52. Assume you are solving the Traveling Salesperson Problem for 4 cities (A, B, C, D) with known distances between each pair of cities. Now, you need to add a fifth city (E) to the problem.

Test Cases

1. Symmetric Distances

• **Description:** All distances are symmetric (distance from A to B is the same as B to A).

Distances:

A-B: 10, A-C: 15, A-D: 20, A-E: 25 B-C: 35, B-D: 25, B-E: 30 C-D: 30, C-E: 20 D-E: 15

Expected Output: The shortest route and its total distance.

Aim: Find the shortest route that visits each city exactly once and returns to the starting city.

Algorithm:

step1: Start at a random city and repeatedly choose the closest unvisited city until all cities have been visited.

step2: Improve the route by swapping two edges if it results in a shorter total distance.

step3: Repeat steps 1-2 until no further improvements can be made.

program:

```
1 import numpy as np
2 def traveling_salesperson(distances):
3     n = len(distances)
4     route = [0]
5     unvisited = set(range(1, n))
6     while unvisited:
7         current = route[-1]
8         next_city = min(unvisited, key=lambda city: distances[current][city])
9         route.append(next_city)
10        unvisited.remove(next_city)
11    for i in range(n):
12        for j in range(i+1, n):
13            new_route = route[:i] + list(reversed(route[i:j])) + route[j:]
14            if sum(distances[new_route[k]][new_route[k+1]] for k in range(n-1)) < sum
               (distances[route[k]][route[k+1]] for k in range(n-1)):
15                route = new_route
16    return route
17 distances = np.array([
18     [0, 10, 15, 20, 25],
19     [10, 0, 35, 25, 30],
20     [15, 35, 0, 30, 20],
21     [20, 25, 30, 0, 15],
22     [25, 30, 20, 15, 0]
23 ])
24 route = traveling_salesperson(distances)
25 print("Shortest Route:", route)
26 print("Total Distance:", sum(distances[route[k]][route[k+1]] for k in range(len(route)-1)))
```

Input:

```
distances = np.array([
    [0, 10, 15, 20, 25],
    [10, 0, 35, 25, 30],
    [15, 35, 0, 30, 20],
    [20, 25, 30, 0, 15],
    [25, 30, 20, 15, 0]
])
```

Output:

```
Shortest Route: [1, 0, 3, 4, 2]
Total Distance: 65
```

Time Complexity: $T(n) = O(n^3)$

Result: The Shortest Route is [0, 1, 3, 4, 2, 0] and the Total Distance: 105

53. Given a string s , return the longest palindromic substring in S .

Example 1:

Input: $s = \text{"babad"}$

Output: "bab" Explanation: "aba" is also a valid answer.

Example 2:

Input: $s = \text{"cbbd"}$

Output: "bb"

Constraints: ● $1 \leq s.length \leq 1000$ ● s consist of only digits and English letters.

Aim: To find the longest palindromic substring in a given string s .

Algorithm:

step1: Expand Around the Center: For each character in the string, consider it as the center of a potential palindrome.

step2: Check for Palindrome: While expanding, check if the characters on both sides of the center are equal. If they are, continue expanding. If not, stop and move to the next character.

step3: Keep Track of the Longest Palindrome, and return the value

Program:

```
def longest_palindrome(s: str) -> str:
    def expand_around_center(s: str, left: int, right: int) -> str:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return s[left + 1:right]
    longest = ""
    for i in range(len(s)):
        palindrome = expand_around_center(s, i, i)
        if len(palindrome) > len(longest):
            longest = palindrome
        palindrome = expand_around_center(s, i, i + 1)
        if len(palindrome) > len(longest):
            longest = palindrome
    return longest
s = "babad"
print(longest_palindrome(s))
```

Input:

```
s = "babad"
```

Output:

```
bab
```

Time complexity: $T(n) = O(n^2)$

Result: The longest palindrome in the string is bab.

54. Given a string *s*, find the length of the longest substring without repeating characters.

Example 1: Input: *s* = "abcabcbb" Output: 3

Explanation: The answer is "abc", with the length of 3.

Example 2: Input: *s* = "bbbbbb" Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3: Input: *s* = "pwwkew" Output: 3

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints: ● $0 \leq s.length \leq 5 * 10^4$ ● *s* consists of English letters, digits, symbols and spaces.

Aim: Find the length of the longest substring without repeating characters in a given string.

Algorithm:

step1: Initialize two pointers: We will use two pointers, start and end, to represent the start and end of the current substring.

step2: Expand the substring: We will expand the substring by moving the end pointer to the right. For each new character, we will check if it is already in the set.

step3: Shrink the substring: If we encounter a repeating character, we will shrink the substring by moving the start pointer to the right until the repeating character is removed from the set.

Program:

```
def length_of_longest_substring(s: str) -> int:
    char_set = set()
    start = 0
    max_length = 0
    for end in range(len(s)):
        while s[end] in char_set:
            char_set.remove(s[start])
            start += 1
        char_set.add(s[end])
        max_length = max(max_length, end - start + 1)
    return max_length
```

Input:

```
s = "abcabcbb"
```

Output:

```
3
```

Time complexity: $T(n)=O(n)$

Result: The program correctly finds the length of the longest substring without repeating characters in the given string. The time complexity is linear, making it efficient for large inputs.

55. Given a string *s* and a dictionary of strings *wordDict*, return true if *s* can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

Input: *s* = "leetcode", *wordDict* = ["leet", "code"]

Output: true

Explanation: Return true because "leetcode" can be segmented as "leet code".

Example 2:

Input: *s* = "applepenapple", *wordDict* = ["apple", "pen"]

Output: true

Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".

Aim: Determine if a given string can be segmented into a space-separated sequence of one or more dictionary words.

Algorithm:

step1: Create a dynamic programming table: We will create a table *dp* where *dp[i]* is True if the string *s[:i]* can be segmented into dictionary words.

step2: Fill the table: We will iterate over the string *s* and for each position *i*, we will check if the substring *s[:i]* can be segmented into dictionary words.

step3: Return the result: We will return *dp[-1]*.

Program:


```
def word_break(s: str, wordDict: list[str]) -> bool:
    dp = [False] * (len(s) + 1)
    dp[0] = True
    for i in range(1, len(s) + 1):
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                break
    return dp[-1]
```

Input:

```
s = "leetcode", wordDict = ["leet", "code"]
```

Output:

```
True
```

Time complexity: $T(n) = O(n^2)$

Result: The program correctly determines whether the input string can be segmented into dictionary words. The time complexity is quadratic, which may not be efficient for very large inputs.

56. Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words. Consider the following dictionary { i, like, sam, sung, samsung, mobile, ice, cream, icecream, man, go, mango }

Input: ilike

Output: Yes

The string can be segmented as "i like".

Input: ilikesamsung

Output: Yes The string can be segmented as "i like samsung" or "i like sam sung".

Aim: Given an input string and a dictionary of words, determine if the input string can be segmented into a space-separated sequence of dictionary words.

Algorithm:

Step 1: Create a set of dictionary words for efficient lookups.

Step 2: Create a dynamic programming table dp of size n+1, where n is the length of the input string. Initialize dp[0] to True, indicating that an empty string can always be segmented.

Step 3: Iterate through the input string and for each position i, check if the substring from 0 to i can be segmented into dictionary words. If it can, mark dp[i+1] as True.

Step 4: Return dp[n], which indicates whether the entire input string can be segmented into dictionary words.

Program:

```
main.py
1 def word_break(s, word_dict):
2     word_set = set(word_dict)
3     n = len(s)
4     dp = [False] * (n + 1)
5     dp[0] = True
6     for i in range(n):
7         if dp[i]:
8             for j in range(i + 1, n + 1):
9                 if s[i:j] in word_set:
10                     dp[j] = True
11     return dp[n]
12 word_dict = ["i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream",
13             "icecream", "man", "go", "mango"]
14 input_str = "ilike"
15 result = word_break(input_str, word_dict)
16 print(result)
```

Input:

```
word_dict = ["i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream",
             "icecream", "man", "go", "mango"]
input_str = "ilike"
```

Output:

```
Output
True

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

Time Complexity: $T(n)=O(n)$

Result:

The output of the example usage is True, indicating that the input string "ilike" can be segmented into a space-separated sequence of dictionary words.

57. Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified. You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters. Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right. For the last line of text, it should be left-justified, and no extra space is inserted between words. A word is defined as a character sequence consisting of non-space characters only. Each word's length is guaranteed to be greater than 0 and not exceed maxWidth. The input array words contains at least one word.

Example 1:

Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth =

16

Output:

```
[ "This is an",  
  "example of text",  
  "justification. "  
]
```

Example 2:

Input: words = ["What", "must", "be", "acknowledgment", "shall", "be"], maxWidth = 16

Output:

```
[  
  "What must be",  
  "acknowledgment ",  
  "shall be "  
]
```

Explanation: Note that the last line is "shall be " instead of "shall be", because the last line must be left-justified instead of fully-justified.

Note that the second line is also left-justified because it contains only one word.

Aim: Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified.

Algorithm:





Step-1: Initialize an empty list result to store the formatted lines.

Step-2: Initialize an empty list current_line to store the words in the current line.

Step-3: Initialize a variable current_width to store the total width of the words in the current line.

Step-4: If adding the current word to the current line would exceed the maxWidth.

Program:

```
main.py    Share  Run

1 def fullJustify(words, maxWidth):
2     result, current_line, current_width = [], [], 0
3     for word in words:
4         if current_width + len(word) + len(current_line) > maxWidth:
5             result.append(justify_line(current_line, maxWidth))
6             current_line, current_width = [], 0
7             current_line.append(word)
8             current_width += len(word)
9     result.append(' '.join(current_line).ljust(maxWidth))
10    return result
11
12 def justify_line(words, maxWidth):
13     total_width, num_spaces, num_gaps = sum(len(word) for word in words),
14         maxWidth - total_width, len(words) - 1
15     if num_gaps == 0:
16         return words[0] + ' ' * num_spaces
17     base_spaces, extra_spaces = num_spaces // num_gaps, num_spaces % num_gaps
18     return ' '.join(word + ' ' * (base_spaces + (i < extra_spaces)) for i, word
19         in enumerate(words)).rstrip() + ' '
20
21 words = ["This", "is", "an", "example", "of", "text", "justification."]
22 maxWidth = 16
23 result = fullJustify(words, maxWidth)
24 print(result)
```

Input:

```
words = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth = 16
```

Output:

```
Output

['This  is  an', 'example of text', 'justification. ']

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

Time Complexity: $T(n) = O(n)$

Result: The program Has been Executed Successfully.

58.Design a special dictionary that searches the words in it by a prefix and a suffix. Implement the WordFilter class: WordFilter(string[] words) Initializes the object with the words in the dictionary.f(string pref, string suff) Returns the index of the word in the dictionary, which has the prefix pref and the suffix suff. If there is more than one valid index, return the largest of them. If there is no such word in the dictionary, return -1.

Example 1:

Input

```
["WordFilter", "f"]  
[[["apple"]], ["a", "e"]]
```

Output[null, 0]

Explanation

WordFilter wordFilter = new WordFilter(["apple"]);
wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = "e".

Aim: The aim of the WordFilter class is to provide an efficient data structure for querying a dictionary of words based on both prefix and suffix criteria.

Algorithm:

step1: Initialize an empty dictionary prefix_suffix_map to store (prefix, suffix) pairs as keys and their corresponding indices as values.

step2: Compute the length of the current word, denoted as L.

step3: Iterate over all possible lengths for prefixes and suffixes.

Program:

```
class WordFilter:  
    def __init__(self, words):  
        self.prefix_suffix_map = {}  
        for index, word in enumerate(words):  
            word_length = len(word)  
            for i in range(word_length + 1):  
                prefix = word[:i]  
                for j in range(word_length + 1):  
                    suffix = word[-j:] if j > 0 else ''  
                    self.prefix_suffix_map[(prefix, suffix)] = index  
    def f(self, pref, suff):  
        return self.prefix_suffix_map.get((pref, suff), -1)  
wordFilter = WordFilter(["apple"])  
print(wordFilter.f("a", "e"))
```

Input:

```
wordFilter = WordFilter(["apple"])  
print(wordFilter.f("a", "e"))
```

Output:

Output

0

Time complexity: $O(n * m^2)$.

Result: The output for the query wordFilter.f("a", "e") is 0

