NAME: D.K.Sumalatha

REG NO: 192424023

COURSE CODE:CSA0613

COURSE NAME: Design And Analysis of Algorithm for Optimal Application

LAB EXPERIMENTS

TOPIC 1

1.Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward.

Aim

To find and return the **first palindromic string** from a given array of strings. If no palindrome exists, return an empty string "".

Algorithm

1. Start.
2. Read the list of words.
3. For each word in the list:
    - Reverse the word.
    - Check if the word is equal to its reverse.
4. If a palindrome is found, return it immediately.
5. If no palindrome is found after checking all words, return an empty string "".
6. Stop.

Code (Python)

```
def first_palindrome(words):
    for word in words:
        if word == word[::-1]:
            return word
    return ""

# Example 1
words1 = ["abc", "car", "ada", "racecar", "cool"]
print(first_palindrome(words1))

# Example 2
words2 = ["notapalindrome", "racecar"]
print(first_palindrome(words2))
```

Input

Example 1:

["abc", "car", "ada", "racecar", "cool"]

Example 2:

["notapalindrome", "racecar"]

Output

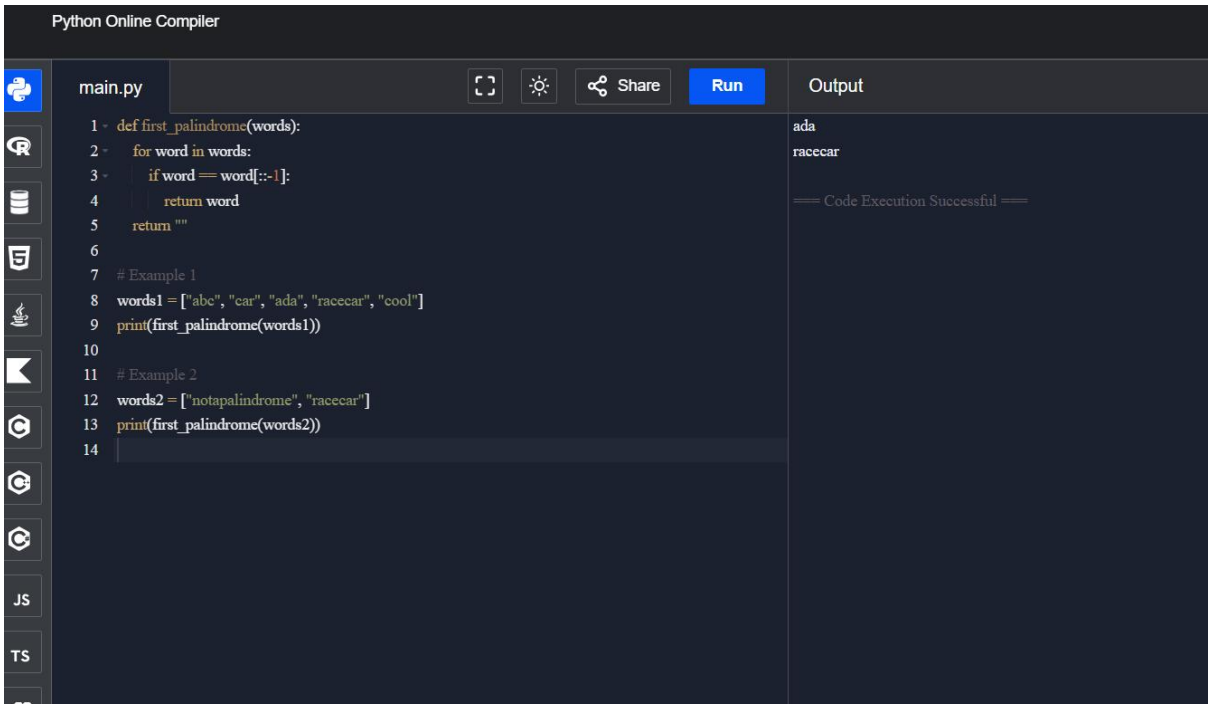
Example 1:

ada

Example 2:

racecar

Implementation



Result

Thus, we are successfully got the output

2.You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1 : the number of indices i such that nums1[i] exists in nums2. answer2 : the number of indices i such that nums2[i] exists in nums1 Return [answer1,answer2].

Aim

To find the number of indices in nums1 whose elements exist in nums2 and the number of indices in nums2 whose elements exist in nums1.

Algorithm

1. Start
2. Read arrays nums1 and nums2
3. Convert nums1 into set1 and nums2 into set2
4. Initialize answer1 = 0 and answer2 = 0
5. For each element in nums1, if it exists in set2, increment answer1
6. For each element in nums2, if it exists in set1, increment answer2

7. Return [answer1, answer2]
8. Stop

Code (Python)

```
def count_common(nums1, nums2):
    set1 = set(nums1)
    set2 = set(nums2)

    answer1 = 0
    answer2 = 0

    for num in nums1:
        if num in set2:
            answer1 += 1

    for num in nums2:
        if num in set1:
            answer2 += 1

    return [answer1, answer2]

# Example 1
nums1 = [2, 3, 2]
nums2 = [1, 2]
print(count_common(nums1, nums2))

# Example 2
nums1 = [4, 3, 2, 3, 1]
nums2 = [2, 2, 5, 2, 3, 6]
print(count_common(nums1, nums2))
```

Input

Example 1:

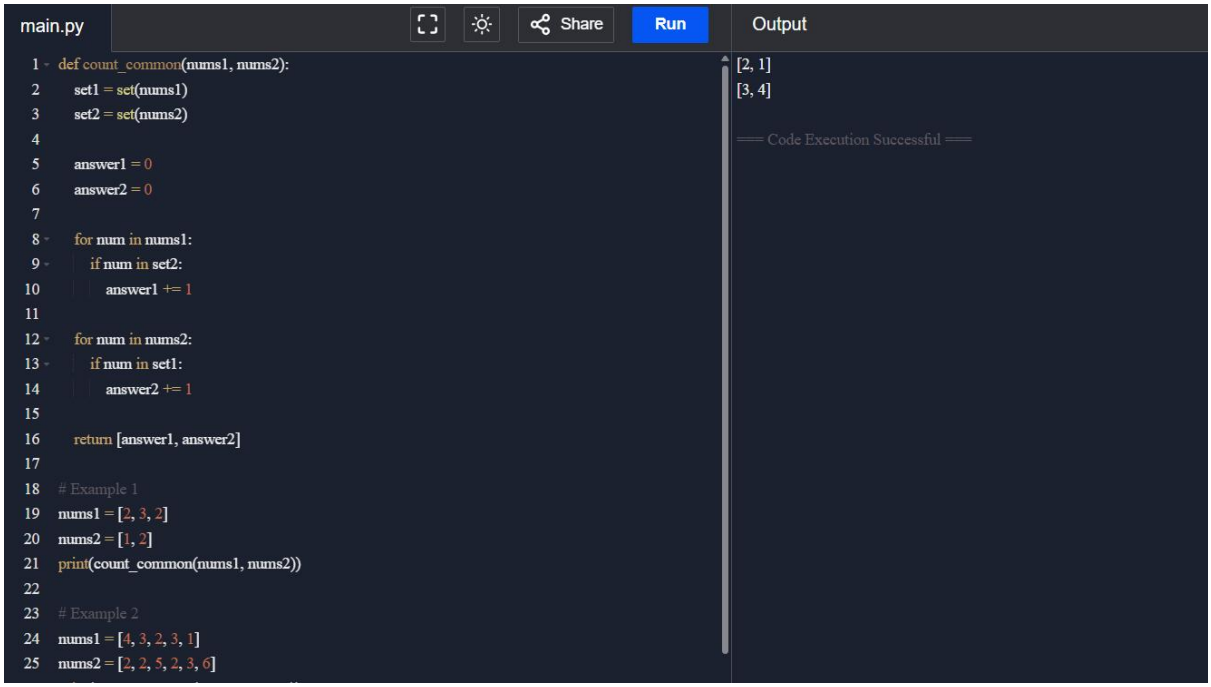
nums1 = [2, 3, 2]
nums2 = [1, 2]

Example 2:
nums1 = [4, 3, 2, 3, 1]
nums2 = [2, 2, 5, 2, 3, 6]

Output
Example 1:
[2, 1]

Example 2:
[3, 4]

Implementation



Result

Thus, we are successfully got the output

3.You are given a 0-indexed integer array nums. The distinct count of a subarray of nums is defined as: Let nums[i..j] be a subarray of nums consisting of all the indices from i to j such that 0 <= i <= j < nums.length. Then the number of distinct values in nums[i..j] is called the distinct count of nums[i..j]. Return the sum of the squares of distinct counts of all subarrays of nums. A subarray is a contiguous non-empty sequence of elements within an array.

Aim

To find the sum of the squares of the distinct counts of all possible contiguous subarrays of a given integer array.

Algorithm

1. Start
2. Read the array nums
3. Initialize result = 0
4. For each starting index i from 0 to n-1
5. Create an empty set to store distinct elements
6. For each ending index j from i to n-1
7. Add nums[j] to the set
8. Find the size of the set (distinct count)
9. Square the distinct count and add it to result
10. Repeat until all subarrays are processed
11. Print the result
12. Stop

Code (Python)

```
def sum_of_squares_of_distinct(nums):
    n = len(nums)
    result = 0

    for i in range(n):
        distinct_set = set()
        for j in range(i, n):
            distinct_set.add(nums[j])
            count = len(distinct_set)
            result += count * count

    return result
```

Example

nums = [1, 2, 1]

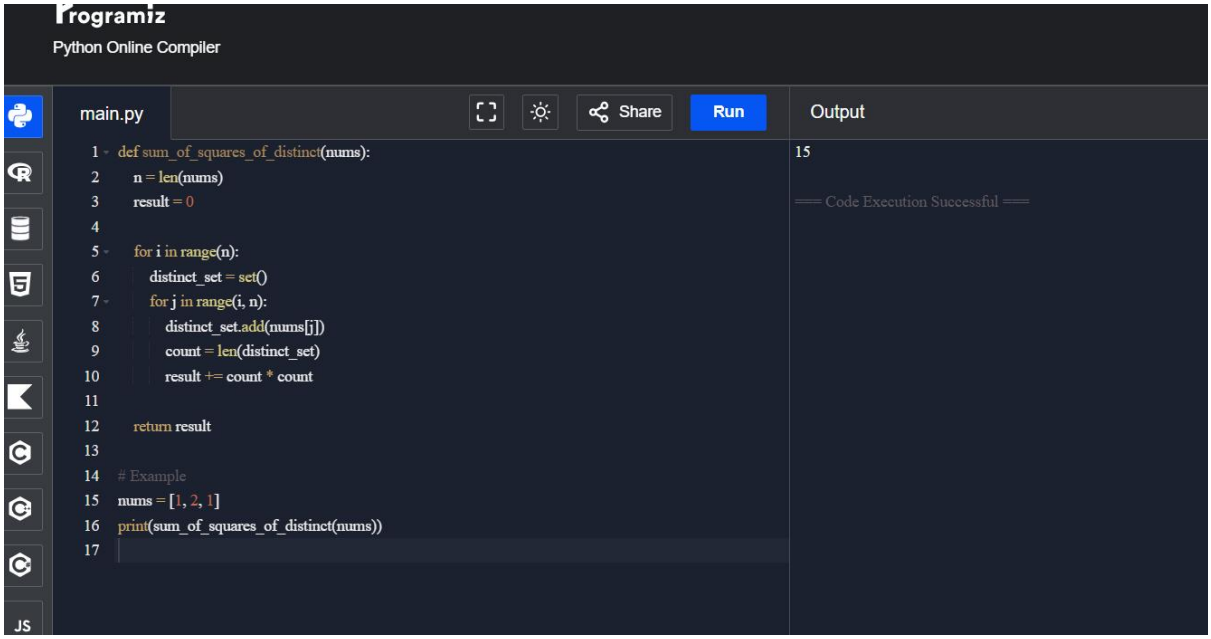
print(sum_of_squares_of_distinct(nums))

Input

nums = [1, 2, 1]

Output

15



Result

Thus, we are successfully got the output

4. Given a 0-indexed integer array nums of length n and an integer k, return the number of pairs (i, j) where 0 <= i < j < n, such that nums[i] == nums[j] and (i * j) is divisible by k.

Aim

To find the number of pairs (i, j) such that nums[i] = nums[j] and the product (i × j) is divisible by k.

Algorithm

1. Start
2. Read the array nums and integer k
3. Initialize count = 0
4. Loop through index i from 0 to n-1
5. Loop through index j from i+1 to n-1
6. If nums[i] == nums[j] and (i * j) % k == 0, increment count
7. Continue until all pairs are checked
8. Return count
9. Stop

Code (Python)

```
def count_pairs(nums, k):
    n = len(nums)
    count = 0

    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] == nums[j] and (i * j) % k == 0:
                count += 1

    return count

# Example 1
nums1 = [3,1,2,2,2,1,3]
k1 = 2
print(count_pairs(nums1, k1))

# Example 2
nums2 = [1,2,3,4]
k2 = 1
```

print(count_pairs(nums2, k2))

Input

Example 1:
nums = [3,1,2,2,2,1,3]
k = 2

Example 2:
nums = [1,2,3,4]
k = 1

Output

Example 1:
4

Example 2:
0

Implementation

```
def count_pairs(nums, k):
    n = len(nums)
    count = 0

    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] == nums[j] and (i * j) % k == 0:
                count += 1

    return count

# Example 1
nums1 = [3,1,2,2,2,1,3]
k1 = 2
print(count_pairs(nums1, k1))

# Example 2
nums2 = [1,2,3,4]
k2 = 1
print(count_pairs(nums2, k2))
```

```
4
0

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

Result

Thus, we are successfully got the output

5. Write a program FOR THE BELOW TEST CASES with least time complexity

Test Cases: -

Input: {1, 2, 3, 4, 5} Expected Output: 5

Input: {7, 7, 7, 7, 7} Expected Output: 7

Input: {-10, 2, 3, -4, 5} Expected Output: 5

Aim

To find the maximum element from a given array using the least time complexity.

Algorithm

1. Start
2. Read the array nums
3. Initialize max_val as the first element of the array
4. Traverse through the array from index 1 to n-1
5. If the current element is greater than max_val, update max_val
6. Continue until the end of the array
7. Print max_val
8. Stop

Code (Python)

```
def find_max(nums):
    max_val = nums[0]
    for num in nums:
        if num > max_val:
            max_val = num
    return max_val


# Test cases
print(find_max([1, 2, 3, 4, 5]))
print(find_max([7, 7, 7, 7, 7]))
print(find_max([-10, 2, 3, -4, 5]))
```

Input

Test Case 1:
{1, 2, 3, 4, 5}

Test Case 2:
{7, 7, 7, 7, 7}

Test Case 3:
{-10, 2, 3, -4, 5}

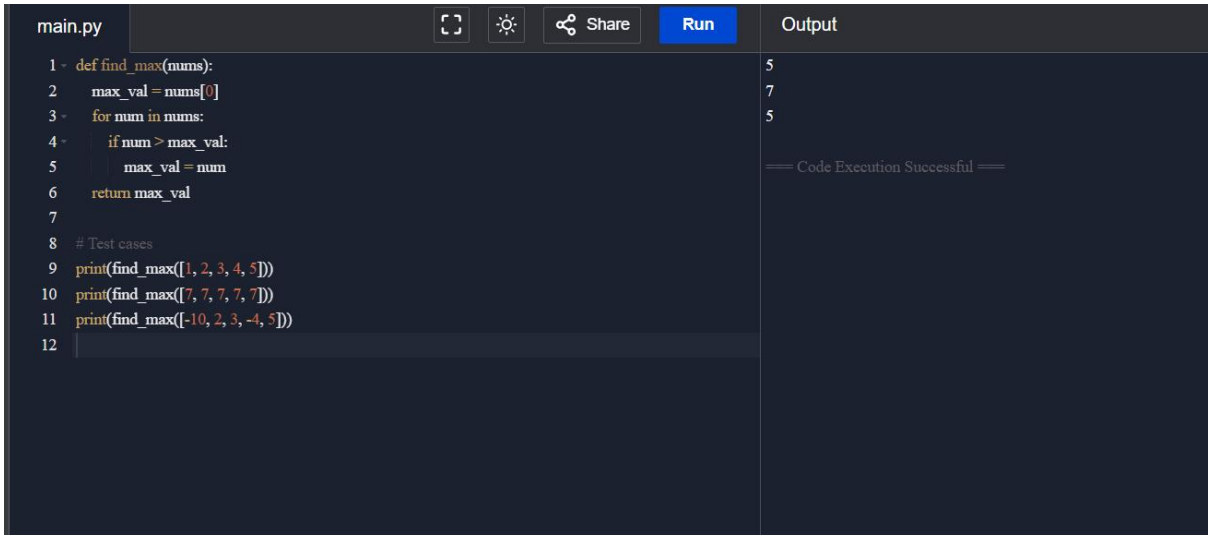
Output

Test Case 1:
5

Test Case 2:
7

Test Case 3:
5

Implementation



Result

Thus, we are successfully got the output

6. You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and then finds the maximum element in sorted list. Write the code for the same.

Test Cases

1. Empty List

1. Input: []

2. Expected Output: None or an appropriate message indicating that the list is empty.

2. Single Element List

1. Input: [5]

2. Expected Output: 5

3. All Elements are the Same

1. Input: [3, 3, 3, 3, 3]

2. Expected Output: 3

Aim

To sort a given list using an efficient sorting method and then find the maximum element from the sorted list.

Algorithm

1. Start
2. Read the input list nums
3. If the list is empty, print an appropriate message and stop
4. Sort the list in ascending order
5. The maximum element will be the last element of the sorted list
6. Print the maximum element
7. Stop

Code (Python)

```
def find_max_after_sort(nums):
    if len(nums) == 0:
        return "List is empty"

    nums.sort()  # Efficient built-in sorting (Timsort)
    return nums[-1]

# Test cases
print(find_max_after_sort([]))            # Empty list
print(find_max_after_sort([5]))           # Single element
```

print(find_max_after_sort([3, 3, 3, 3, 3])) # All elements same

Input

Test Case 1:
[]

Test Case 2:
[5]

Test Case 3:
[3, 3, 3, 3, 3]

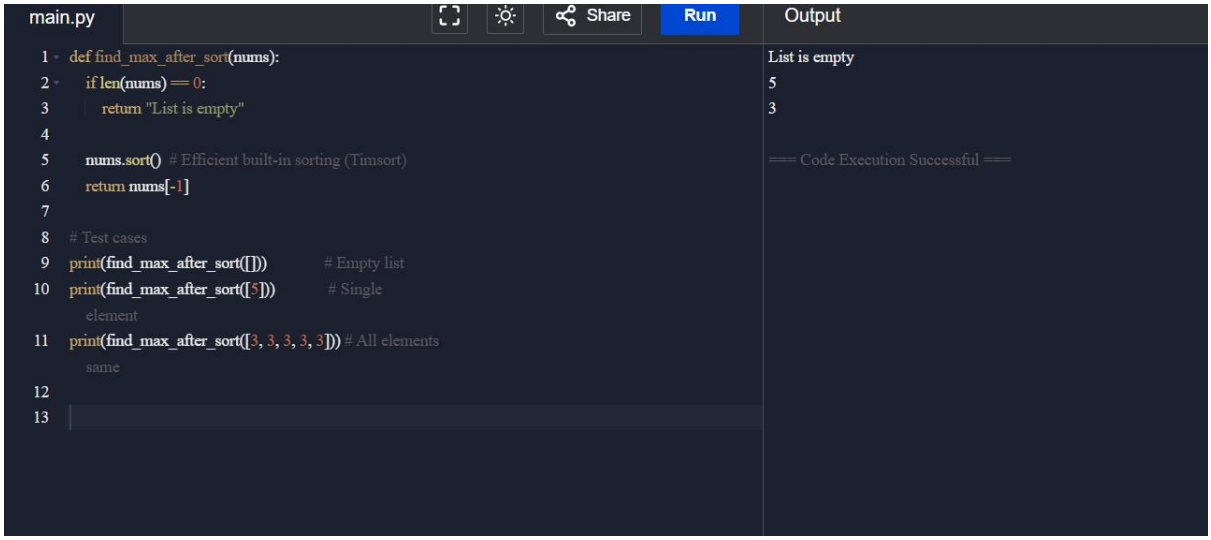
Output

Test Case 1:
List is empty

Test Case 2:
5

Test Case 3:
3

Implementation



Result

Thus, we are successfully got the output

7. Write a program that takes an input list of n numbers and creates a new list containing only the unique elements from the original list. What is the space complexity of the algorithm?

Test Cases

Some Duplicate Elements

Input: [3, 7, 3, 5, 2, 5, 9, 2]

Expected Output: [3, 7, 5, 2, 9] (Order may vary based on the algorithm used)

Negative and Positive Numbers

Input: [-1, 2, -1, 3, 2, -2]

Expected Output: [-1, 2, 3, -2] (Order may vary)

List with Large Numbers

Input: [1000000, 999999, 1000000]

Expected Output: [1000000, 999999]

Aim

To create a new list containing only the unique elements from a given list of n numbers and determine the space complexity of the algorithm.

Algorithm

1. Start
2. Read the input list nums
3. Create an empty set called seen
4. Create an empty list called unique_list
5. Traverse through each element in nums
6. If the element is not in seen, add it to seen and append it to unique_list
7. Continue until all elements are processed
8. Print unique_list
9. Stop

Code (Python)

```
def get_unique_elements(nums):
    seen = set()
    unique_list = []

    for num in nums:
```

```
        if num not in seen:
            seen.add(num)
            unique_list.append(num)

    return unique_list

# Test cases
print(get_unique_elements([3, 7, 3, 5, 2, 5, 9, 2]))
print(get_unique_elements([-1, 2, -1, 3, 2, -2]))
print(get_unique_elements([1000000, 999999, 1000000]))
```

Input

Test Case 1:
[3, 7, 3, 5, 2, 5, 9, 2]

Test Case 2:
[-1, 2, -1, 3, 2, -2]

Test Case 3:
[1000000, 999999, 1000000]

Output

Test Case 1:
[3, 7, 5, 2, 9]

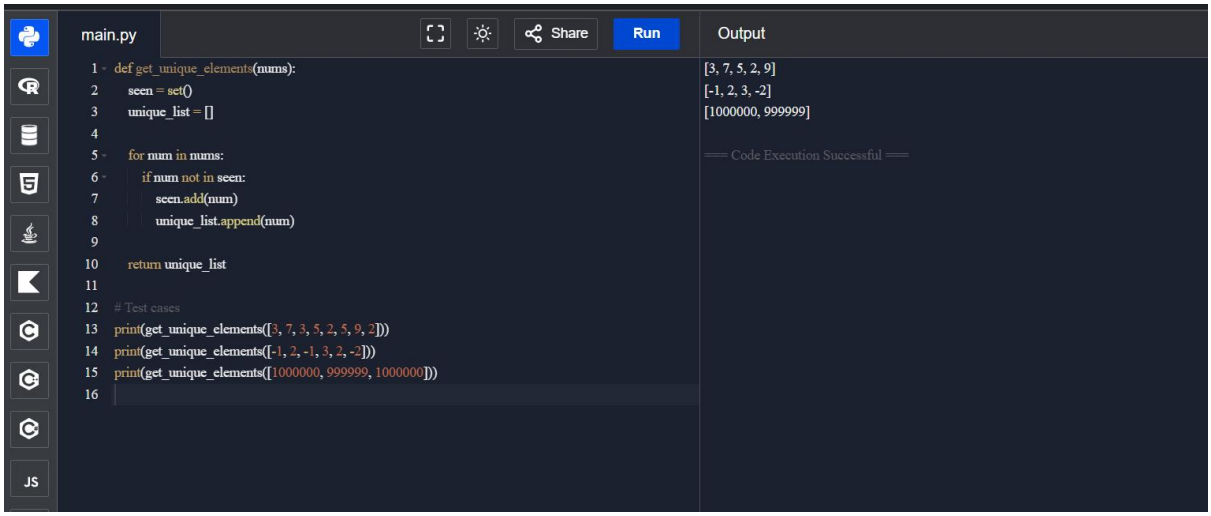
Test Case 2:
[-1, 2, 3, -2]

Test Case 3:
[1000000, 999999]

Space Complexity

The space complexity of this algorithm is **O(n)** because an additional set and list are used to store up to n unique elements.

Implementation



Result

Thus, we are successfully got the output

8. Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code

Aim

To sort an array of integers using the Bubble Sort technique and analyze its time complexity using Big-O notation.

Algorithm

1. Start
2. Read the input array nums
3. Let n be the length of the array
4. Repeat the following steps for i from 0 to n-1
5. For each i, compare adjacent elements from index 0 to n-i-2
6. If the current element is greater than the next element, swap them
7. Continue until the array is sorted
8. Print the sorted array
9. Stop

Code (Python)

```
def bubble_sort(nums):
    n = len(nums)
    for i in range(n):
```

```
        for j in range(0, n - i - 1):
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
    return nums


# Example
arr = [5, 1, 4, 2, 8]
print(bubble_sort(arr))
```

Input

[5, 1, 4, 2, 8]

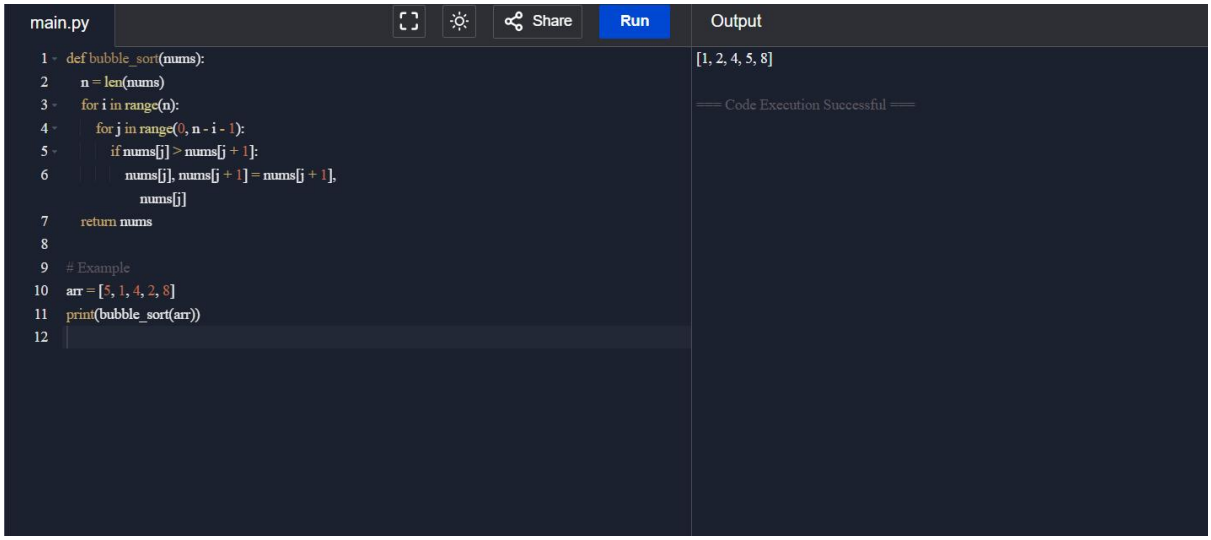
Output

[1, 2, 4, 5, 8]

Time Complexity (Big-O Analysis)

Worst Case: $O(n^2)$ (when the array is sorted in reverse order)
Average Case: $O(n^2)$
Best Case: O(n) (when the array is already sorted, with optimization)

Implementation



Result

Thus, we are successfully got the output

9. Checks if a given number x exists in a sorted array arr using binary search.

Analyze its time complexity using Big-O notation.

Test Case:

Example X={ 3,4,6,-9,10,8,9,30} KEY=10

Output: Element 10 is found at position 5

Example X={ 3,4,6,-9,10,8,9,30} KEY=100

Output : Element 100 is not found

Aim

To check whether a given number x exists in a sorted array using the Binary Search technique and analyze its time complexity using Big-O notation.

Algorithm

1. Start
2. Read the array arr and the key x
3. Sort the array (since Binary Search works only on sorted arrays)
4. Set low = 0 and high = len(arr) - 1
5. While low <= high
6. Find mid = (low + high) // 2
7. If arr[mid] == x, print the position and stop
8. If arr[mid] < x, set low = mid + 1
9. Else, set high = mid - 1
10. If the loop ends, print that the element is not found
11. Stop

Code (Python)

```
def binary_search(arr, x):
    arr.sort()  # Sorting the array first
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
```

```
        if arr[mid] == x:
            return f"Element {x} is found at position {mid + 1}"
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1

    return f"Element {x} is not found"

# Test cases
arr1 = [3, 4, 6, -9, 10, 8, 9, 30]
key1 = 10
print(binary_search(arr1, key1))

arr2 = [3, 4, 6, -9, 10, 8, 9, 30]
key2 = 100
print(binary_search(arr2, key2))
```

Input

Example 1:
X = {3, 4, 6, -9, 10, 8, 9, 30}
KEY = 10

Example 2:
X = {3, 4, 6, -9, 10, 8, 9, 30}
KEY = 100

Output

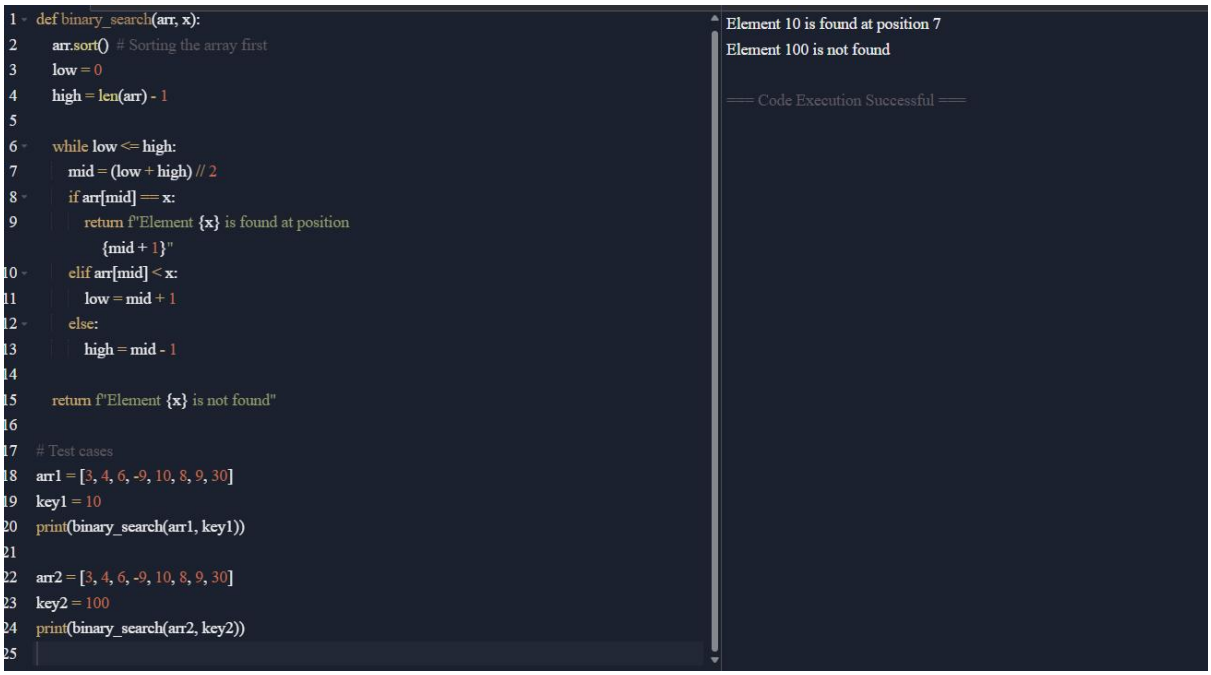
Example 1:
Element 10 is found at position 5

Example 2:
Element 100 is not found

Time Complexity (Big-O Analysis)

Best Case: O(1)
Average Case: O(log n)
Worst Case: O(log n)

Implementation



```
def binary_search(arr, x):
    arr.sort()  # Sorting the array first
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return f"Element {x} is found at position
                {mid + 1}"
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1

    return f"Element {x} is not found"

# Test cases
arr1 = [3, 4, 6, -9, 10, 8, 9, 30]
key1 = 10
print(binary_search(arr1, key1))

arr2 = [3, 4, 6, -9, 10, 8, 9, 30]
key2 = 100
print(binary_search(arr2, key2))
```

```
Element 10 is found at position 7
Element 100 is not found

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

Result

Thus, we are successfully got the output

10. Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

Aim

To sort an array of integers in ascending order without using any built-in functions, achieving a time complexity of **O(n log n)** and minimum possible space complexity.

Algorithm (Merge Sort)

1. Start
2. Read the input array nums
3. If the length of the array is less than or equal to 1, return the array
4. Divide the array into two halves

5. Recursively apply merge sort on both halves
6. Merge the two sorted halves into a single sorted array
7. Return the sorted array
8. Stop

Code (Python)

```
def merge_sort(nums):
    if len(nums) <= 1:
        return nums

    mid = len(nums) // 2
    left = merge_sort(nums[:mid])
    right = merge_sort(nums[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    while i < len(left):
        result.append(left[i])
        i += 1
```

```
    while j < len(right):
        result.append(right[j])
        j += 1

    return result

# Example
nums = [5, 2, 3, 1]
sorted_nums = merge_sort(nums)
print(sorted_nums)
```

Input

nums = [5, 2, 3, 1]

Output

[1, 2, 3, 5]

Time Complexity (Big-O Analysis)

Best Case: O(n log n)
Average Case: O(n log n)
Worst Case: O(n log n)

Space Complexity

O(n) due to the temporary arrays used during the merge process.

Implementation

```python
def merge_sort(nums):
    if len(nums) <= 1:
        return nums

    mid = len(nums) // 2
    left = merge_sort(nums[:mid])
    right = merge_sort(nums[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    while i < len(left):
        result.append(left[i])
        i += 1
```

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

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

Result

Thus, we are successfully got the output

11. Given an m x n grid and a ball at a starting cell, find the number of ways to move the ball out of the grid boundary in exactly N steps.

Example:

· Input: m=2,n=2,N=2,i=0,j=0 · Output: 6

· Input: m=1,n=3,N=3,i=0,j=1 · Output: 12

Aim

To find the number of ways to move a ball out of an m × n grid boundary in exactly N steps starting from a given cell (i, j).

Algorithm (Dynamic Programming)

1. Start
2. Read values m, n, N, i, j
3. Create a 2D DP array dp of size m × n initialized to 0
4. Set dp[i][j] = 1 (starting position)
5. Initialize count = 0
6. For each step from 1 to N
7. Create a new 2D array temp of size m × n initialized to 0
8. For each cell (r, c) in the grid

9. Try moving up, down, left, and right
10. If the move goes out of the grid, add dp[r][c] to count
11. Else, add dp[r][c] to the corresponding new position in temp
12. Replace dp with temp
13. After N steps, return count
14. Stop

Code (Python)

```
def find_paths(m, n, N, i, j):
    dp = [[0 for _ in range(n)] for _ in range(m)]
    dp[i][j] = 1
    count = 0


    for _ in range(N):
        temp = [[0 for _ in range(n)] for _ in range(m)]
        for r in range(m):
            for c in range(n):
                if dp[r][c] > 0:
                    val = dp[r][c]
                    # Up
                    if r - 1 < 0:
                        count += val
                    else:
                        temp[r - 1][c] += val
                    # Down
                    if r + 1 >= m:
                        count += val
                    else:
                        temp[r + 1][c] += val
                    # Left
                    if c - 1 < 0:
```

```
                count += val
            else:
                temp[r][c - 1] += val
            # Right
            if c + 1 >= n:
                count += val
            else:
                temp[r][c + 1] += val
    dp = temp

return count

# Test cases
print(find_paths(2, 2, 2, 0, 0))
print(find_paths(1, 3, 3, 0, 1))
```

Input

Example 1:
m = 2, n = 2, N = 2, i = 0, j = 0

Example 2:
m = 1, n = 3, N = 3, i = 0, j = 1

Output

Example 1:
6

Example 2:
12

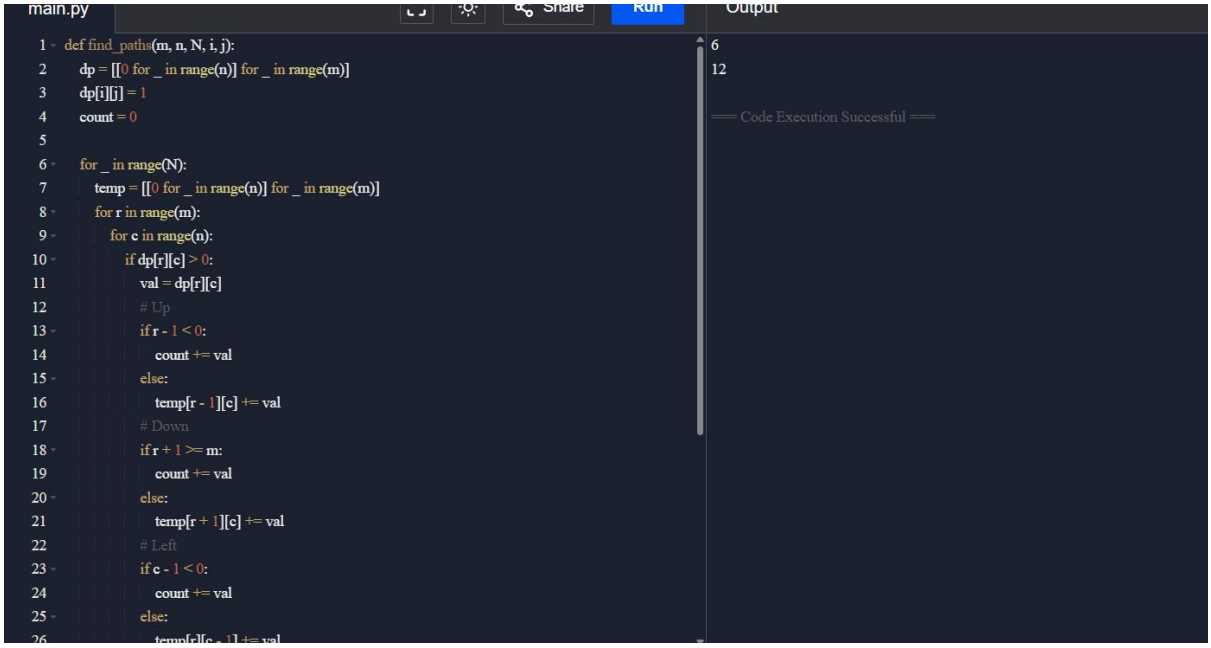
Time Complexity

O(N × m × n)

Space Complexity

O(m × n)

Implementation



Result

Thus, we are successfully got the output

12. You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security systems connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.

Examples:

Input : nums = [2, 3, 2]

Output : The maximum money you can rob without alerting the police is 3(robbing house 1).

(ii)Input : nums = [1, 2, 3, 1]

Output : The maximum money you can rob without alerting the police is 4 (robbing house 1 and house 3).

Aim

To determine the maximum amount of money that can be robbed from houses arranged in a circular manner without robbing two adjacent houses.

Algorithm (Dynamic Programming)

1. Start
2. Read the input array nums
3. If there is only one house, return its value
4. Since houses are in a circle, solve two cases:
    - Case 1: Rob houses from index 0 to n-2
    - Case 2: Rob houses from index 1 to n-1
5. Use a helper function to find the maximum sum for each case using DP
6. Return the maximum of the two cases
7. Stop

Code (Python)

```
def rob(nums):
    if len(nums) == 1:
        return nums[0]
    return max(rob_linear(nums[:-1]), rob_linear(nums[1:]))
def rob_linear(arr):
    prev1 = 0
    prev2 = 0
    for num in arr:
        temp = prev1
        prev1 = max(prev2 + num, prev1)
        prev2 = temp
    return prev1


# Test cases
print(rob([2, 3, 2]))
print(rob([1, 2, 3, 1]))
```

Input

Example 1:
nums = [2, 3, 2]

Example 2:
nums = [1, 2, 3, 1]

Output

Example 1:
The maximum money you can rob without alerting the police is 3

Example 2:
The maximum money you can rob without alerting the police is 4

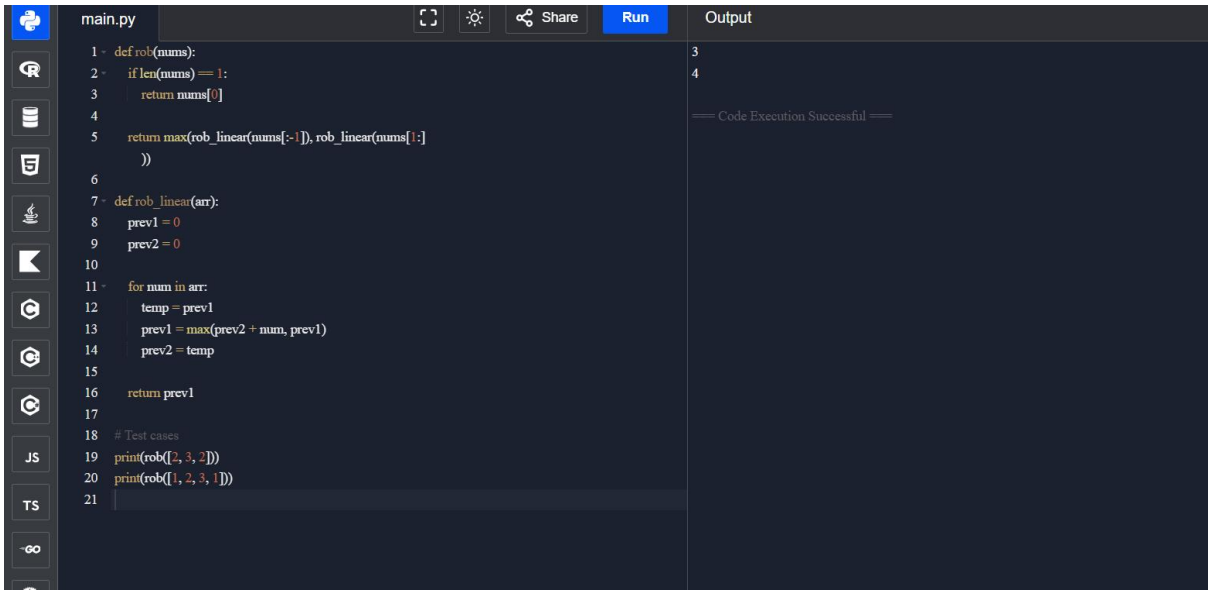
Time Complexity

O(n)

Space Complexity

O(1)

Implementation



Result

Thus, we are successfully got the output

13. You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Examples:

Input: n=4 Output: 5

Input: n=3 Output: 3

Aim

To find the number of distinct ways to climb a staircase of n steps when you can take either 1 step or 2 steps at a time.

Algorithm (Dynamic Programming)

1. Start
2. Read the value of n
3. If n is 0 or 1, return 1
4. Create an array dp of size n+1
5. Set dp[0] = 1 and dp[1] = 1
6. For i from 2 to n
7. Set dp[i] = dp[i-1] + dp[i-2]
8. Return dp[n]
9. Stop

Code (Python)

```
def climb_stairs(n):
    if n == 0 or n == 1:
        return 1

    dp = [0] * (n + 1)
    dp[0] = 1
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

# Test cases
```

print(climb_stairs(4))

print(climb_stairs(3))

Test cases

print(climb_stairs(4))

print(climb_stairs(3))

Input

Example 1:
n = 4

Example 2:
n = 3

Output

Example 1:
5

Example 2:
3

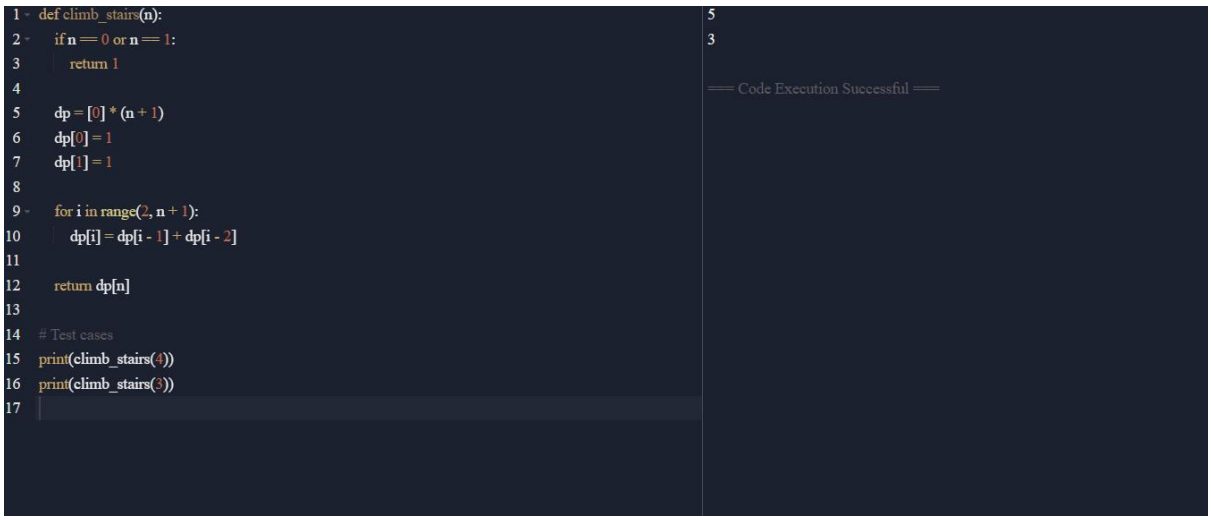
Time Complexity

O(n)

Space Complexity

O(n)

Implementation



```
def climb_stairs(n):
    if n == 0 or n == 1:
        return 1

    dp = [0] * (n + 1)
    dp[0] = 1
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

# Test cases
print(climb_stairs(4))
print(climb_stairs(3))
```

```
5
3

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

Result

Thus, we are successfully got the output

14. A robot is located at the top-left corner of a m×n grid .The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid. How many possible unique paths are there?

Examples:

Input: m=7,n=3 Output: 28

Input: m=3,n=2 Output: 3

Aim

To find the number of unique paths a robot can take to move from the top-left corner to the bottom-right corner of an m × n grid, moving only right or down.

Algorithm (Dynamic Programming)

1. Start
2. Read the values m and n
3. Create a 2D array dp of size m × n
4. Initialize the first row and first column with 1 (only one way to move)
5. For each cell dp[i][j], calculate dp[i][j] = dp[i-1][j] + dp[i][j-1]
6. Continue until the bottom-right corner is reached
7. Return dp[m-1][n-1]
8. Stop

Code (Python)

```
def unique_paths(m, n):
    dp = [[0 for _ in range(n)] for _ in range(m)]

    for i in range(m):
        dp[i][0] = 1
    for j in range(n):
        dp[0][j] = 1

    for i in range(1, m):
```

```
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

    return dp[m - 1][n - 1]

# Test cases
print(unique_paths(7, 3))
print(unique_paths(3, 2))
```

Input

Example 1:
m = 7, n = 3

Example 2:
m = 3, n = 2

Output

Example 1:
28

Example 2:
3

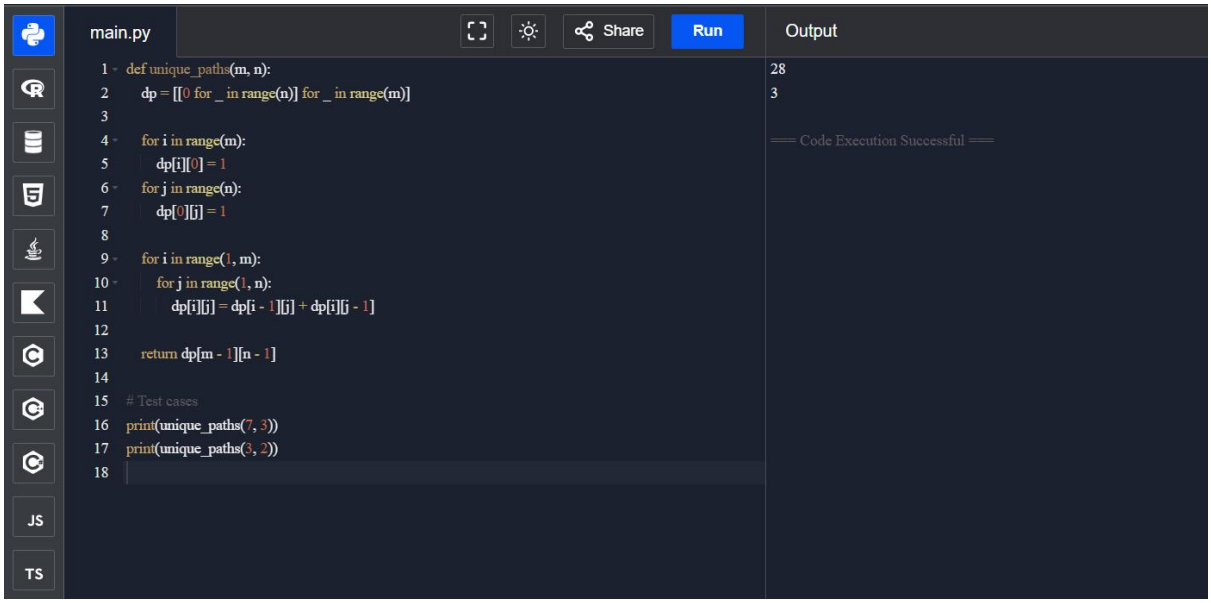
Time Complexity

O(m × n)

Space Complexity

O(m × n)

Implementation



Result

Thus, we are successfully got the output

15. In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a string like s = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z", and "yy". A group is identified by an interval [start, end], where start and end denote the start and end indices (inclusive) of the group.
In the above example, "xxxx" has the interval [3,6]. A group is considered large if it has 3 or more characters. Return the intervals of every large group sorted in increasing order by start index.

Aim

To identify all large groups (3 or more consecutive identical characters) in a string and return their intervals [start, end] sorted by start index.

Algorithm

1. Start
2. Read the input string s
3. Initialize result as an empty list
4. Initialize start = 0
5. Traverse the string with index i from 1 to len(s)
6. If s[i] != s[i-1] or i == len(s)

- Check if the group length i - start >= 3
- If yes, append [start, i-1] to result
- Update start = i

7. Continue until the end of the string
8. Return result
9. Stop

Code (Python)

```
def large_group_positions(s):
    result = []
    start = 0

    for i in range(1, len(s) + 1):
        if i == len(s) or s[i] != s[i - 1]:
            if i - start >= 3:
                result.append([start, i - 1])
            start = i

    return result

# Test cases
print(large_group_positions("abbxxxxzzy"))
print(large_group_positions("abc"))
```

Input

Example 1:
s = "abbxxxxzzy"

Example 2:
s = "abc"

Output

Example 1:
[[3, 6]]

Example 2:
[]

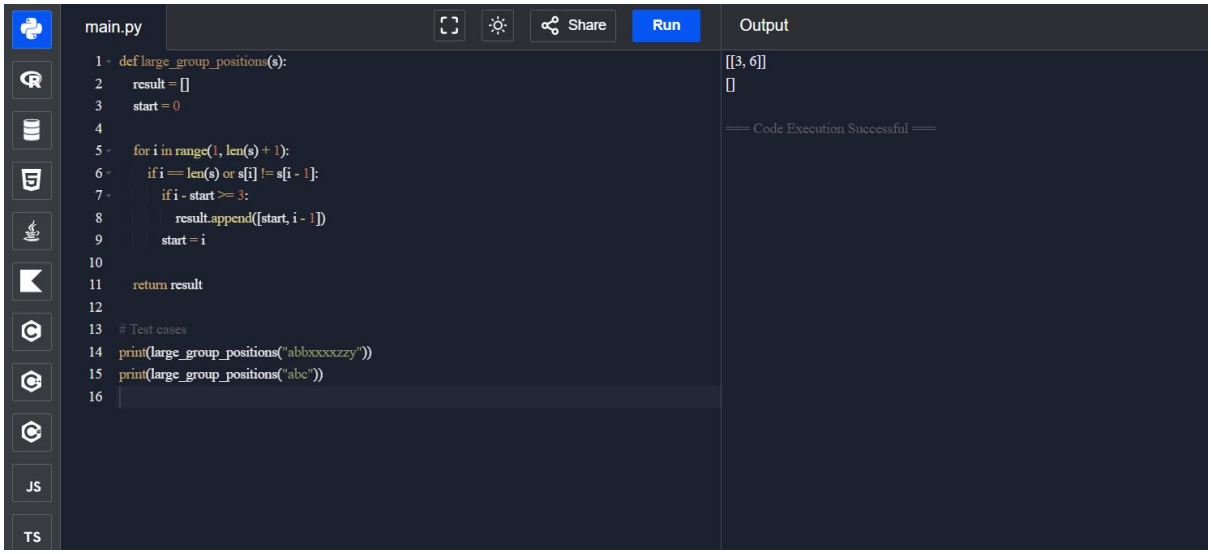
Time Complexity

O(n) where n is the length of the string

Space Complexity

O(k) where k is the number of large groups found

Implementation



Result

Thus, we are successfully got the output

15. "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules

Any live cell with fewer than two live neighbors dies as if caused by under-population.

Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population.

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return the next state.

Aim

To compute the next state of a given m × n grid for Conway's Game of Life by applying the rules of under-population, survival, over-population, and reproduction simultaneously to all cells.

Algorithm

1. Start
2. Read the input grid board of size m × n
3. Create a copy of the board or mark intermediate states to avoid overwriting
4. For each cell at position (i, j)
    - Count the number of live neighbors (8 directions: horizontal, vertical, diagonal)
    - Apply the rules:
        - If live and neighbors < 2 → dies
        - If live and neighbors 2 or 3 → lives
        - If live and neighbors > 3 → dies
        - If dead and neighbors == 3 → becomes live
5. Update the board simultaneously based on the computed next state
6. Return the updated board
7. Stop

Code (Python)

```
def game_of_life(board):
    m, n = len(board), len(board[0])

    # Directions for 8 neighbors
    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1),          (0, 1),
                  (1, -1), (1, 0), (1, 1)]
```

```
    # Copy board to track original states
    copy_board = [[board[i][j] for j in range(n)] for i in range(m)]

    for i in range(m):
        for j in range(n):
            live_neighbors = 0
            for dx, dy in directions:
                ni, nj = i + dx, j + dy
                if 0 <= ni < m and 0 <= nj < n and copy_board[ni][nj] == 1:
                    live_neighbors += 1

            if copy_board[i][j] == 1:
                if live_neighbors < 2 or live_neighbors > 3:
                    board[i][j] = 0
            else:
                if live_neighbors == 3:
                    board[i][j] = 1

    return board

# Example
board = [[0,1,0],
         [0,0,1],
         [1,1,1],
         [0,0,0]]

next_state = game_of_life(board)
for row in next_state:
    print(row)
```

Input

board =
[[0,1,0],
[0,0,1],
[1,1,1],
[0,0,0]]

Output

[[0,0,0],
[1,0,1],
[0,1,1],
[0,1,0]]

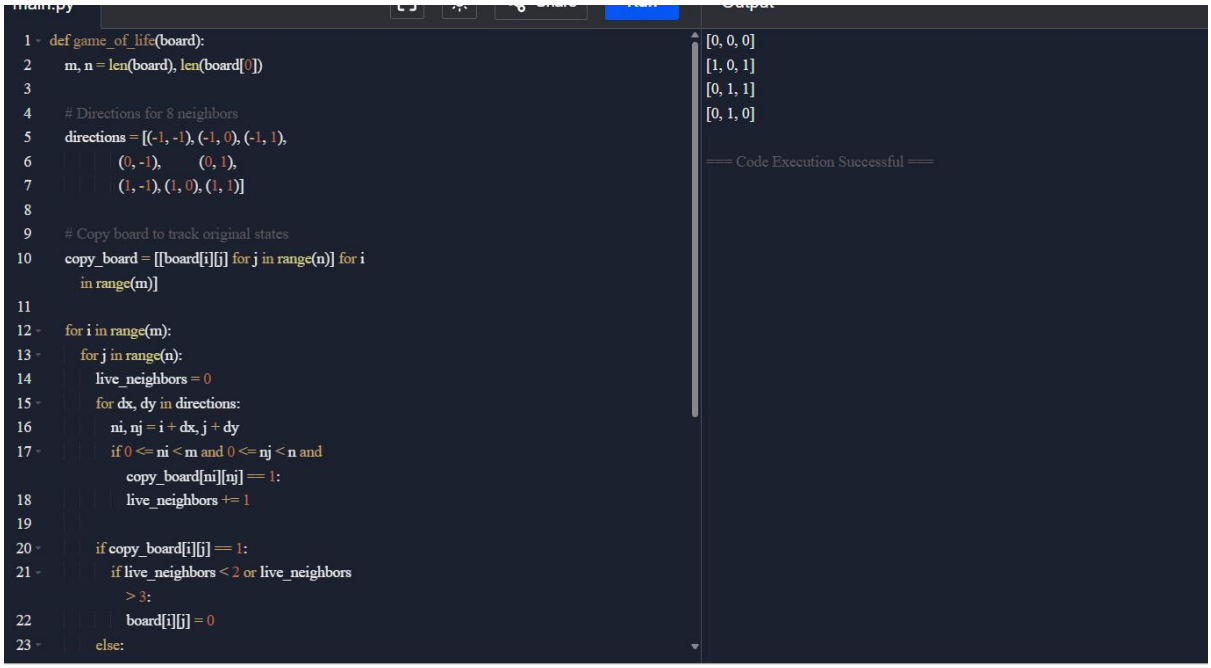
Time Complexity

O(m × n) because each cell and its neighbors are visited once

Space Complexity

O(m × n) if a copy of the board is used; can be optimized to O(1) with in-place encoding

Implementation



```
def game_of_life(board):
    m, n = len(board), len(board[0])

    # Directions for 8 neighbors
    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1),          (0, 1),
                  (1, -1), (1, 0), (1, 1)]

    # Copy board to track original states
    copy_board = [[board[i][j] for j in range(n)] for i
        in range(m)]

    for i in range(m):
        for j in range(n):
            live_neighbors = 0
            for dx, dy in directions:
                ni, nj = i + dx, j + dy
                if 0 <= ni < m and 0 <= nj < n and
                    copy_board[ni][nj] == 1:
                    live_neighbors += 1

            if copy_board[i][j] == 1:
                if live_neighbors < 2 or live_neighbors
                    > 3:
                    board[i][j] = 0
            else:
```

```
[0, 0, 0]
[1, 0, 1]
[0, 1, 1]
[0, 1, 0]

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

Result

Thus, we are successfully got the output

16. We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row. Each glass holds one cup of

champagne. Then, some champagne is poured into the first glass at the top. When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it. When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on. (A glass at the bottom row has its excess champagne fall on the floor.) For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.

Now after pouring some non-negative integer cups of champagne, return how full the jth glass in the ith row is (both i and j are 0-indexed.)

Aim

To determine how full a specific glass is in a champagne tower after pouring a given number of cups, following the rule that overflow from a glass is equally split to the glasses immediately below it.

Algorithm (Dynamic Programming)

1. Start
2. Read input values: poured, query_row, and query_glass
3. Create a 2D array dp with dimensions (query_row+2) × (query_row+2) initialized to 0
4. Pour the champagne into the top glass: dp[0][0] = poured
5. For each row i from 0 to query_row
6. For each glass j in row i
    - If dp[i][j] > 1 (overflow occurs)
    - Calculate excess = dp[i][j] - 1
    - Add half of excess to the glass below-left: dp[i+1][j] += excess / 2

- Add half of excess to the glass below-right: dp[i+1][j+1] += excess / 2
- Set dp[i][j] = 1 (since a glass can hold at most 1 cup)

7. Return min(1, dp[query_row][query_glass]) as the fullness of the requested glass
8. Stop

Code (Python)

```
def champagneTower(poured, query_row, query_glass):
    dp = [[0] * (query_row + 2) for _ in range(query_row + 2)]
    dp[0][0] = poured

    for i in range(query_row + 1):
        for j in range(i + 1):
            if dp[i][j] > 1:
                excess = dp[i][j] - 1
                dp[i+1][j] += excess / 2
                dp[i+1][j+1] += excess / 2
                dp[i][j] = 1

    return dp[query_row][query_glass]

# Test cases
print(champagneTower(1, 1, 1))
print(champagneTower(2, 1, 1))
```

Input

Example 1:
poured = 1, query_row = 1, query_glass = 1

Example 2:
poured = 2, query_row = 1, query_glass = 1

Output

Example 1:
0.0

Example 2:
0.5

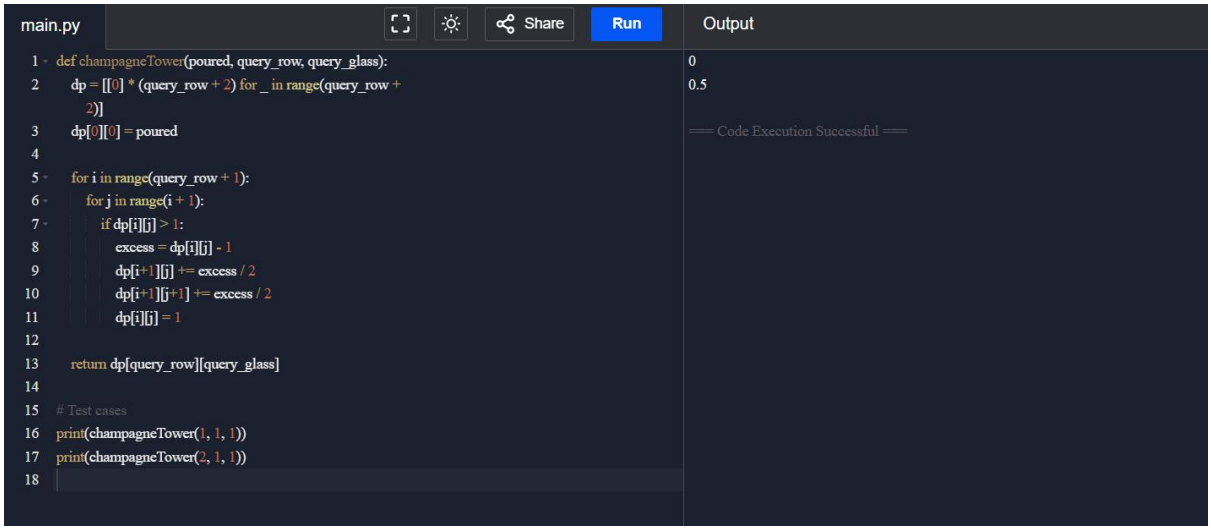
Time Complexity

O(query_row²) because each glass up to the query row is visited once

Space Complexity

O(query_row²) for the DP array

Implementation



Result

Thus, we are successfully got the output

TOPIC 2

1. Write a program to perform the following

An empty list

A list with one element

A list with all identical elements

A list with negative numbers

Aim

To write a program that handles different types of lists (empty, single element, all identical, and with negative numbers) and sorts them when necessary.

Algorithm

1. Start

2. Read the input list nums
3. Check the type of list:
    - If empty, return []
    - If list has one element, return the same list
    - Otherwise, sort the list in ascending order
4. Return the processed list
5. Stop

Code (Python)

```
def process_list(nums):
    if len(nums) == 0:
        return []
    elif len(nums) == 1:
        return nums
    else:
        return sorted(nums)


# Test cases
print(process_list([]))              # Empty list
print(process_list([1]))             # Single element
print(process_list([7, 7, 7, 7]))    # All identical
print(process_list([-5, -1, -3, -2, -4])) # Negative numbers
```

Input

Test Case 1: []

Test Case 2: [1]

Test Case 3: [7, 7, 7, 7]

Test Case 4: [-5, -1, -3, -2, -4]

Output

Test Case 1: []

Test Case 2: [1]

Test Case 3: [7, 7, 7, 7]

Test Case 4: [-5, -4, -3, -2, -1]

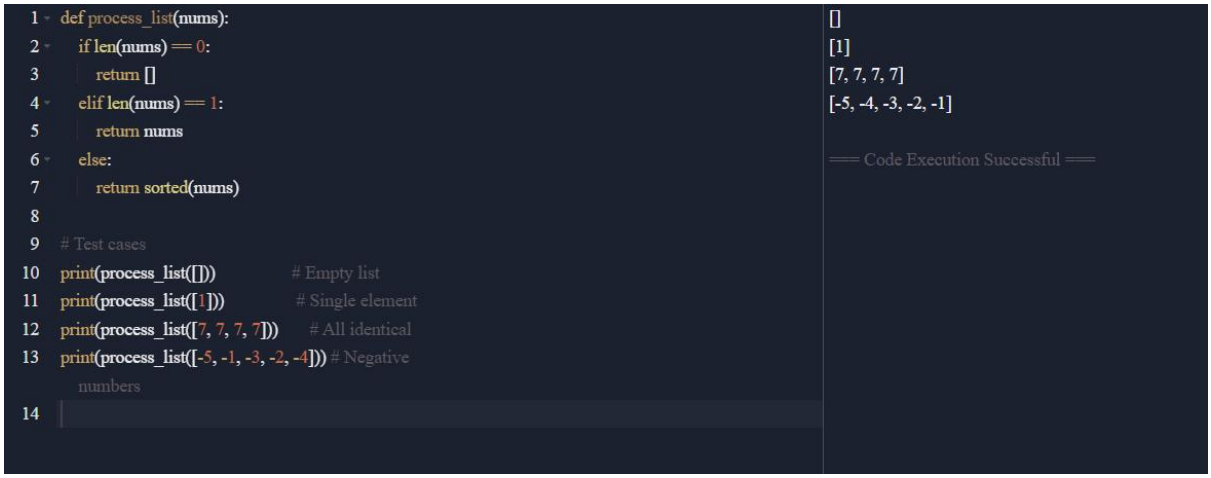
Time Complexity

O(n log n) for sorting the list (for lists with more than one element)

Space Complexity

O(n) for the sorted list

Implementation



```
def process_list(nums):
    if len(nums) == 0:
        return []
    elif len(nums) == 1:
        return nums
    else:
        return sorted(nums)

# Test cases
print(process_list([]))              # Empty list
print(process_list([1]))             # Single element
print(process_list([7, 7, 7, 7]))    # All identical
print(process_list([-5, -1, -3, -2, -4])) # Negative
    numbers

```

```
[]
[1]
[7, 7, 7, 7]
[-5, -4, -3, -2, -1]

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

Result

Thus, we are successfully got the output

2. Describe the Selection Sort algorithm's process of sorting an array. Selection Sort works by dividing the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements. The algorithm repeatedly selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element, then moves the boundary of the sorted region one element to the right. Explain why Selection Sort is simple to understand and implement but is inefficient for large datasets. Provide examples to illustrate step-by-step how Selection Sort rearranges the elements into ascending order, ensuring clarity in your explanation of the algorithm's mechanics and effectiveness.

Aim

To explain and demonstrate the Selection Sort algorithm, showing how it sorts arrays into ascending order and analyzing its simplicity and inefficiency for large datasets.

Algorithm (Selection Sort)

1. Start
2. Divide the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements.
3. For each position i from 0 to n-1:
    - Find the smallest element in the unsorted region (from index i to n-1)
    - Swap this smallest element with the element at index i
    - Move the boundary of the sorted region one element to the right
4. Repeat until the entire array is sorted
5. Stop

Example: Sorting a Random Array [5, 2, 9, 1, 5, 6]

- Initial array: [5, 2, 9, 1, 5, 6]
- Step 1: Find min (1), swap with index 0 → [1, 2, 9, 5, 5, 6]
- Step 2: Find min in remaining [2, 9, 5, 5, 6] is 2, swap with index 1 → [1, 2, 9, 5, 5, 6]
- Step 3: Find min in remaining [9, 5, 5, 6] is 5, swap with index 2 → [1, 2, 5, 9, 5, 6]
- Step 4: Find min in [9, 5, 6] is 5, swap with index 3 → [1, 2, 5, 5, 9, 6]
- Step 5: Find min in [9, 6] is 6, swap with index 4 → [1, 2, 5, 5, 6, 9]
- Step 6: Last element already in place → [1, 2, 5, 5, 6, 9]

Example: Sorting a Reverse Sorted Array [10, 8, 6, 4, 2]

- Step 1: Min = 2, swap with index 0 → [2, 8, 6, 4, 10]
- Step 2: Min = 4, swap with index 1 → [2, 4, 6, 8, 10]
- Step 3: Min = 6, already at index 2 → [2, 4, 6, 8, 10]
- Step 4: Min = 8, already at index 3 → [2, 4, 6, 8, 10]
- Step 5: Last element → [2, 4, 6, 8, 10]

Example: Sorting an Already Sorted Array [1, 2, 3, 4, 5]

- Step 1: Min = 1, already at index 0 → [1, 2, 3, 4, 5]
- Step 2: Min = 2, already at index 1 → [1, 2, 3, 4, 5]
- Step 3: Min = 3, already at index 2 → [1, 2, 3, 4, 5]
- Step 4: Min = 4, already at index 3 → [1, 2, 3, 4, 5]
- Step 5: Last element → [1, 2, 3, 4, 5]

Code (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

# Test cases
print(selection_sort([5, 2, 9, 1, 5, 6]))
print(selection_sort([10, 8, 6, 4, 2]))
print(selection_sort([1, 2, 3, 4, 5]))
```

Input

Random Array: [5, 2, 9, 1, 5, 6]

Reverse Sorted Array: [10, 8, 6, 4, 2]

Already Sorted Array: [1, 2, 3, 4, 5]

Output

Random Array: [1, 2, 5, 5, 6, 9]

Reverse Sorted Array: [2, 4, 6, 8, 10]

Already Sorted Array: [1, 2, 3, 4, 5]

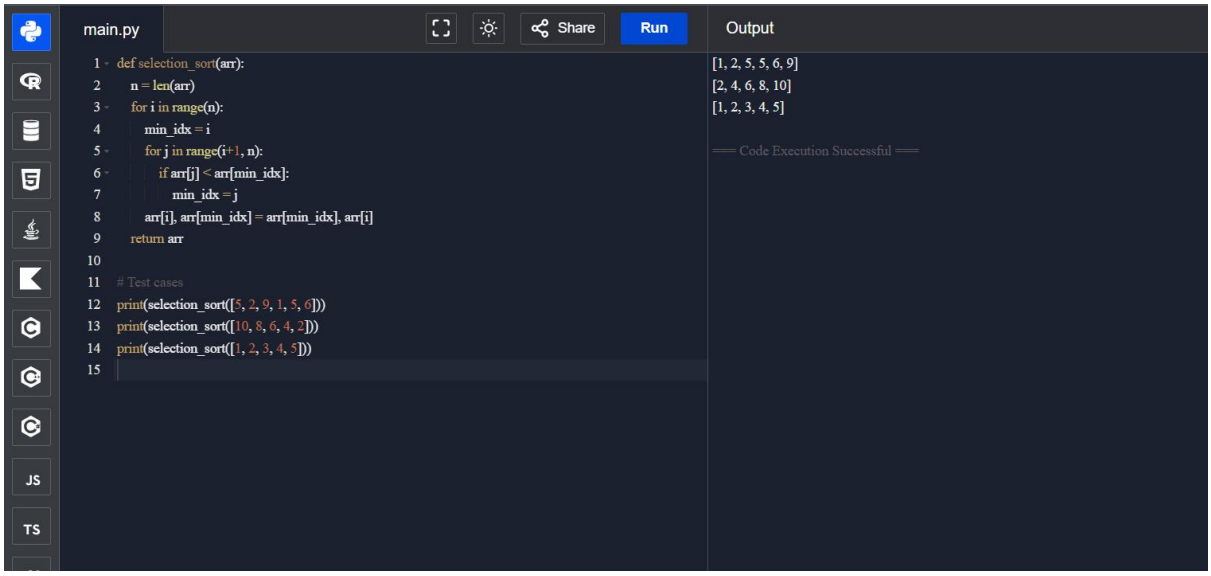
Time Complexity

$O(n^2)$ for all cases (best, average, worst)

Space Complexity

O(1) (in-place sorting)

Implementation



Result

Thus, we are successfully got the output

3. Write code to modify bubble_sort function to stop early if the list becomes sorted before all passes are completed.

Aim

To optimize the Bubble Sort algorithm by stopping early if the list becomes sorted before completing all passes, reducing unnecessary comparisons.

Algorithm (Optimized Bubble Sort)

1. Start
2. Read the input array nums
3. Let n be the length of the array
4. For each pass i from 0 to n-1:
    - Initialize a flag swapped = False
    - For each index j from 0 to n-i-2:
        - If nums[j] > nums[j+1], swap them and set swapped = True
    - If swapped remains False after the inner loop, break the outer loop (array is sorted)
5. Return the sorted array
6. Stop

Code (Python)

```
def bubble_sort_optimized(nums):
    n = len(nums)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
                swapped = True
        if not swapped:
            break
    return nums


# Test cases
print(bubble_sort_optimized([64, 25, 12, 22, 11]))
print(bubble_sort_optimized([29, 10, 14, 37, 13]))
print(bubble_sort_optimized([3, 5, 2, 1, 4]))
print(bubble_sort_optimized([1, 2, 3, 4, 5]))
print(bubble_sort_optimized([5, 4, 3, 2, 1]))
```

Input

1. [64, 25, 12, 22, 11]
2. [29, 10, 14, 37, 13]
3. [3, 5, 2, 1, 4]
4. [1, 2, 3, 4, 5] (Already sorted)
5. [5, 4, 3, 2, 1] (Reverse sorted)

Output

1. [11, 12, 22, 25, 64]
2. [10, 13, 14, 29, 37]
3. [1, 2, 3, 4, 5]
4. [1, 2, 3, 4, 5]
5. [1, 2, 3, 4, 5]

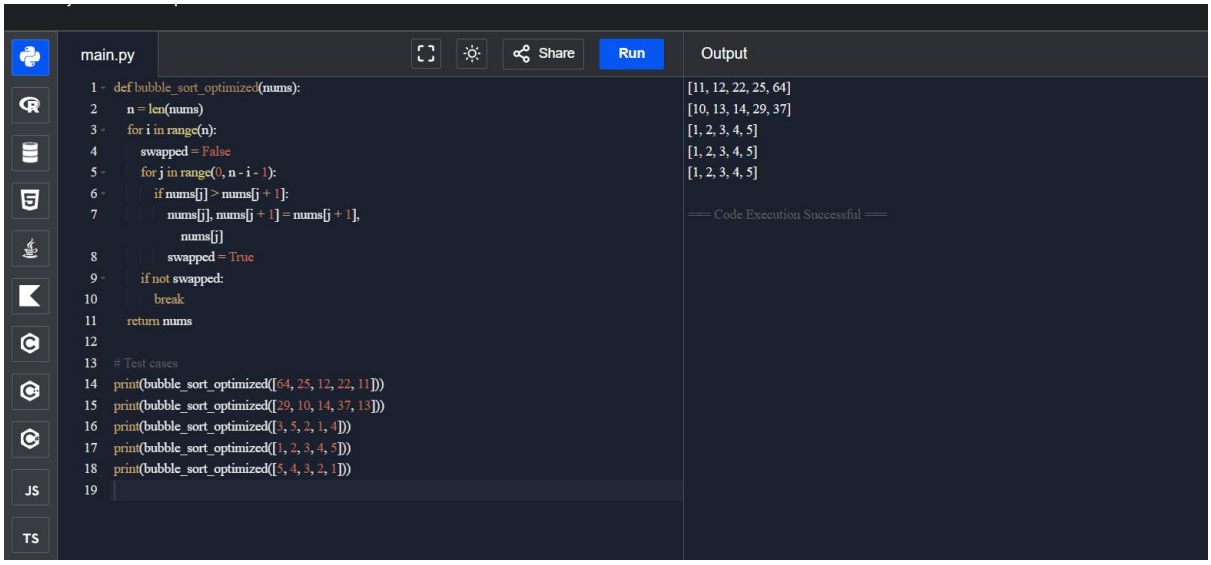
Time Complexity

- Best Case: O(n) when the list is already sorted (early stopping)
- Average/Worst Case: O(n^2)

Space Complexity

O(1) (in-place sorting)

Implementation



Result

Thus, we are successfully got the output

4.Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome.

Aim

To implement Insertion Sort that correctly handles arrays with duplicate elements while preserving their relative order (stable sort), ensuring duplicates are sorted in ascending order.

Algorithm (Insertion Sort with Duplicates)

1. Start
2. Read the input array nums
3. For index i from 1 to n-1:

- Store key = nums[i]
- Initialize j = i - 1
- While j >= 0 and nums[j] > key:
  - Shift nums[j] one position to the right
  - Decrement j
- Place key at position j + 1

4. Continue until all elements are processed
5. Return the sorted array
6. Stop

Note: Insertion Sort is stable, so duplicate elements preserve their relative order.

Code (Python)

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


# Test cases
print(insertion_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3]))  # Array with duplicates
print(insertion_sort([5, 5, 5, 5, 5]))                 # All identical elements
print(insertion_sort([2, 3, 1, 3, 2, 1, 1, 3]))        # Mixed duplicates
```

Input

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

Output

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

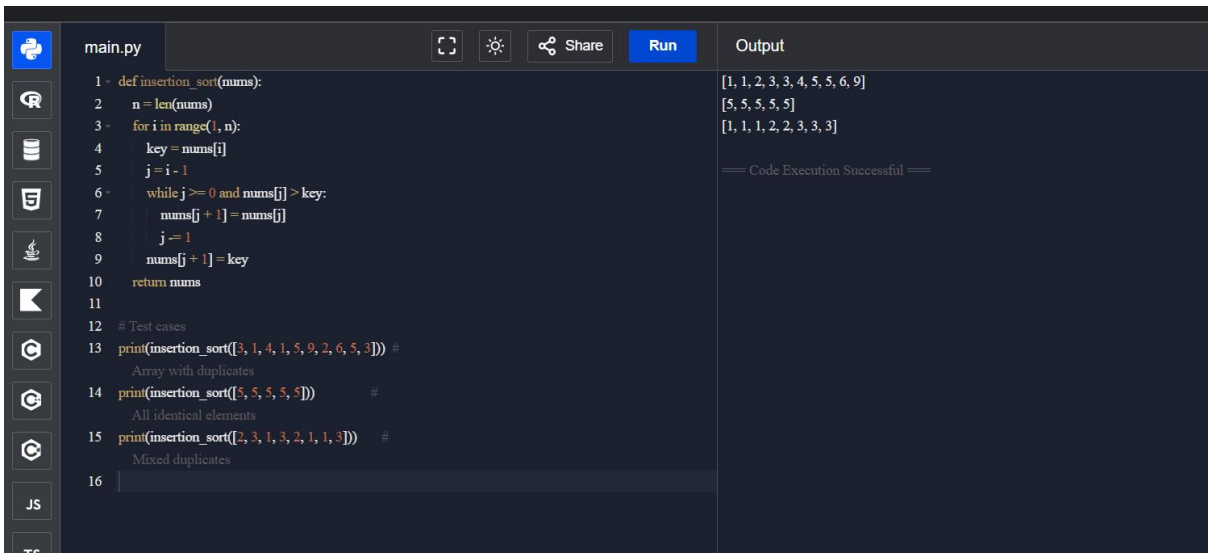
Time Complexity

- Best Case: O(n) when the array is already sorted
- Average/Worst Case: $O(n^2)$

Space Complexity

O(1) (in-place sorting)

Implementation



Result

Thus, we are successfully got the output

5. Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array.

Aim

To find the kth missing positive integer from a sorted array of strictly increasing positive integers.

Algorithm

1. Start
2. Read the input array arr and integer k
3. Initialize missing_count = 0 and current = 1
4. Initialize an index i = 0 to traverse the array
5. Loop until missing_count < k:
    - If i < len(arr) and arr[i] == current:
        - Move to the next element in the array (i += 1)
    - Else:
        - Increment missing_count
        - If missing_count == k, return current
    - Increment current
6. Stop

Code (Python)

```
def find_kth_missing(arr, k):
    missing_count = 0
    current = 1
    i = 0
    n = len(arr)

    while True:
        if i < n and arr[i] == current:
            i += 1
        else:
            missing_count += 1
            if missing_count == k:
                return current
        current += 1

# Test cases
print(find_kth_missing([2, 3, 4, 7, 11], 5))  # Output: 9
```

print(find_kth_missing([1, 2, 3, 4], 2)) # Output: 6

Input

Example 1: arr = [2, 3, 4, 7, 11], k = 5
Example 2: arr = [1, 2, 3, 4], k = 2

Output

Example 1: 9
Example 2: 6

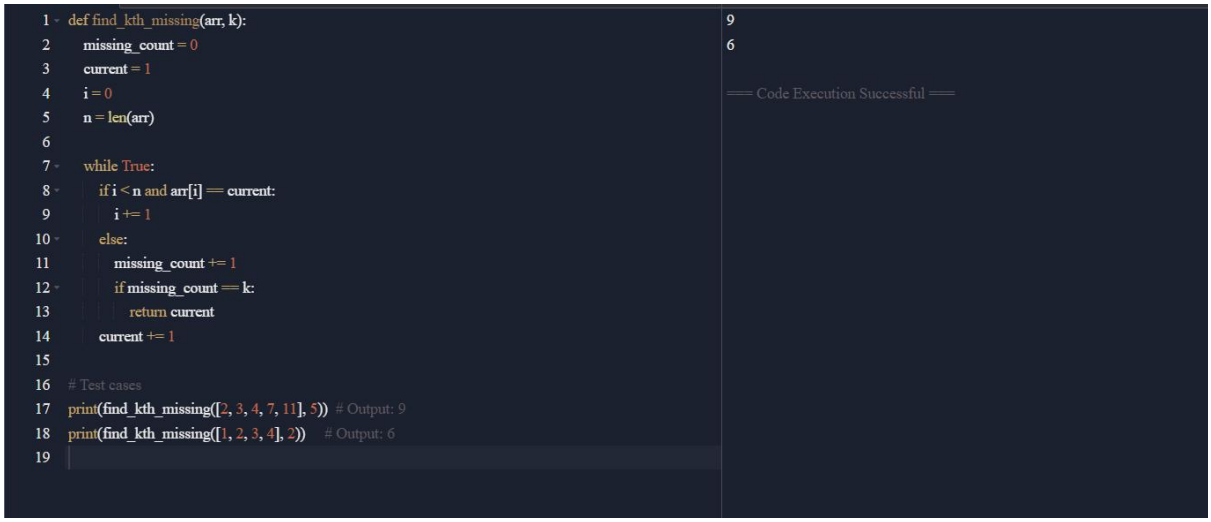
Time Complexity

O(n + k) — we may traverse the array and count missing numbers up to k

Space Complexity

O(1) — constant extra space used

Implementation



```
def find_kth_missing(arr, k):
    missing_count = 0
    current = 1
    i = 0
    n = len(arr)

    while True:
        if i < n and arr[i] == current:
            i += 1
        else:
            missing_count += 1
            if missing_count == k:
                return current
        current += 1

# Test cases
print(find_kth_missing([2, 3, 4, 7, 11], 5))  # Output: 9
print(find_kth_missing([1, 2, 3, 4], 2))  # Output: 6
```

```
9
6

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

Result

Thus, we are successfully got the output

6. A peak element is an element that is strictly greater than its neighbors. Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks. You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array. You must write an algorithm that runs in O(log n) time.

Aim

To find a peak element in an array where a peak is defined as an element strictly greater than its neighbors, using an algorithm with O(log n) time complexity.

Algorithm (Binary Search for Peak Element)

1. Start
2. Read the input array nums
3. Initialize left = 0 and right = len(nums) - 1
4. While left < right:
    - Calculate mid = (left + right) // 2
    - If nums[mid] < nums[mid + 1]:
        - Move left = mid + 1 (peak must be on the right)
    - Else:
        - Move right = mid (peak is at mid or on the left)
5. When left == right, return left as the index of a peak element
6. Stop

Code (Python)

```
def find_peak_element(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = (left + right) // 2
        if nums[mid] < nums[mid + 1]:
            left = mid + 1
        else:
            right = mid
    return left

# Test cases
print(find_peak_element([1, 2, 3, 1]))          # Output: 2
print(find_peak_element([1, 2, 1, 3, 5, 6, 4])) # Output: 1 or 5
```

Input

Example 1: nums = [1, 2, 3, 1]
Example 2: nums = [1, 2, 1, 3, 5, 6, 4]

Output

Example 1: 2
Example 2: 1 or 5

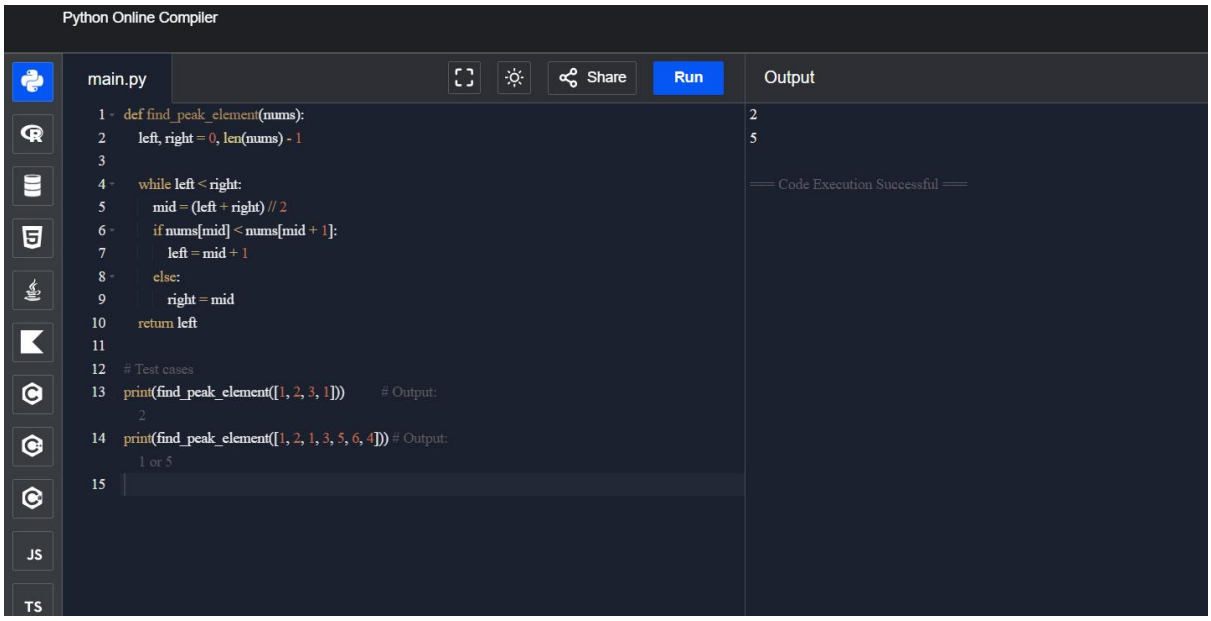
Time Complexity

O(log n) — binary search halves the search space each iteration

Space Complexity

O(1) — constant space, in-place binary search

Implementation



Result

Thus, we are successfully got the output

7. Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Aim

To find the index of the first occurrence of the string needle in the string haystack and return -1 if needle is not found.

Algorithm (Naive Approach)

1. Start

2. Read input strings haystack and needle
3. If needle is empty, return 0
4. Loop over i from 0 to len(haystack) - len(needle):
    - If the substring haystack[i:i+len(needle)] equals needle, return i
5. If the loop completes without finding a match, return -1
6. Stop

Code (Python)

```
def str_str(haystack, needle):
    if not needle:
        return 0

    n, m = len(haystack), len(needle)

    for i in range(n - m + 1):
        if haystack[i:i + m] == needle:
            return i
    return -1

# Test cases
print(str_str("sadbutsad", "sad"))   # Output: 0
print(str_str("leetcode", "leeto"))  # Output: -1
```

Input

Example 1: haystack = "sadbutsad", needle = "sad"
Example 2: haystack = "leetcode", needle = "leeto"

Output

Example 1: 0
Example 2: -1

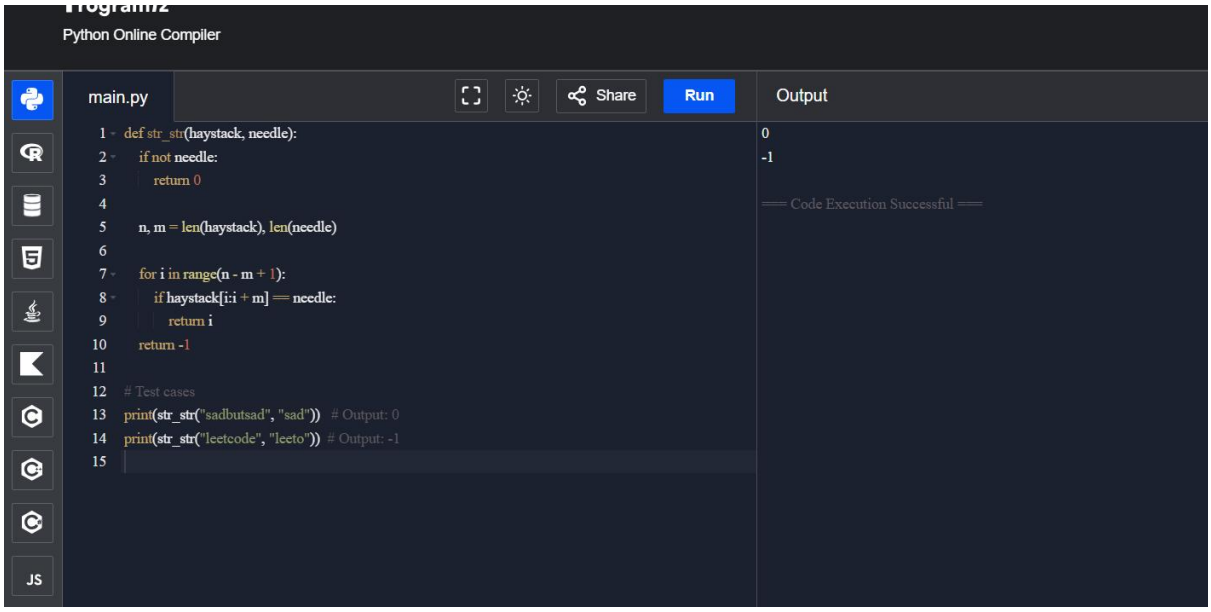
Time Complexity

O((n-m+1) * m) — where n = length of haystack, m = length of needle, because each possible substring is compared

Space Complexity

O(1) — only constant extra space is used

Implementation



Result

Thus, we are successfully got the output

8. Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order. A substring is a contiguous sequence of characters within a string

Aim

To find all strings in a given list that are substrings of at least one other string in the list.

Algorithm

1. Start
2. Read the input list words
3. Initialize an empty list result to store substrings
4. For each word i in words:
    - For each word j in words where i != j:
        - If i is a substring of j, add i to result and break
5. Return result

6. Stop

Code (Python)

```
def string_matching(words):
    result = []
    n = len(words)

    for i in range(n):
        for j in range(n):
            if i != j and words[i] in words[j]:
                result.append(words[i])
                break
    return result

# Test cases
print(string_matching(["mass","as","hero","superhero"]))  # Output: ["as","hero"]
print(string_matching(["leetcode","et","code"]))          # Output: ["et","code"]
print(string_matching(["blue","green","bu"]))             # Output: []
```

Input

Example 1: words = ["mass","as","hero","superhero"]
Example 2: words = ["leetcode","et","code"]
Example 3: words = ["blue","green","bu"]

Output

Example 1: ["as","hero"] (or ["hero","as"])
Example 2: ["et","code"]
Example 3: []

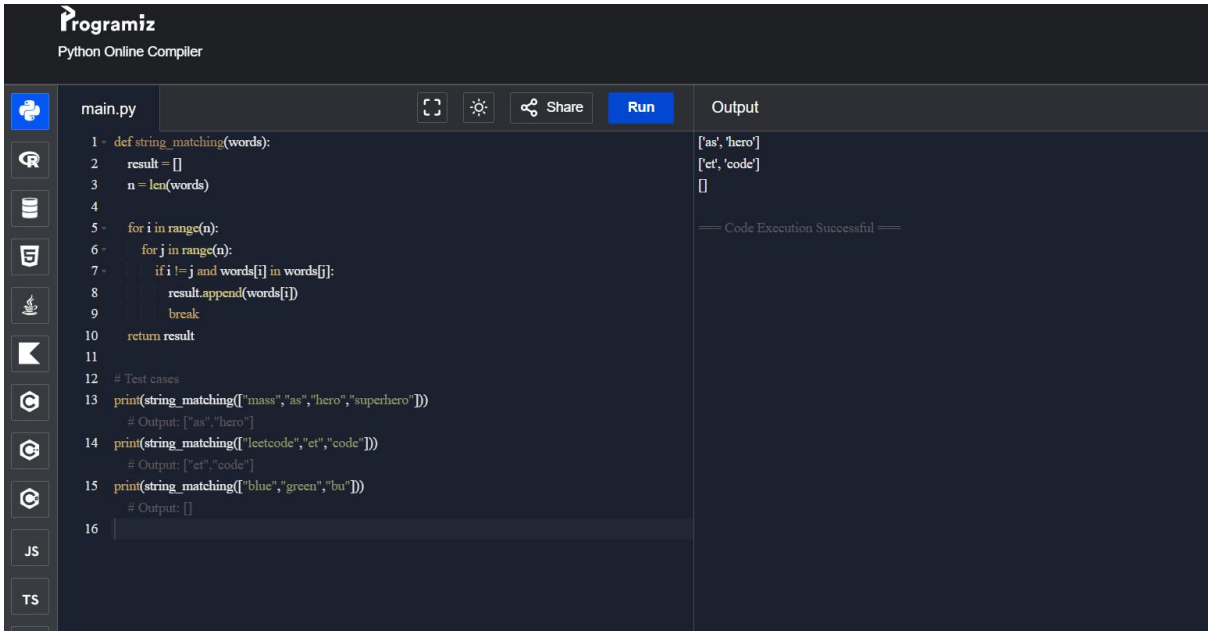
Time Complexity

$O(n^2 * m)$ — n = number of words, m = average length of a word (for substring check)

Space Complexity

O(n) — for storing the result list

Implementation



Result

Thus, we are successfully got the output

9. Write a program that finds the closest pair of points in a set of 2D points using the brute force approach.

Aim

To find the closest pair of points in a set of 2D points using the brute-force approach, calculating the Euclidean distance between all possible pairs.

Algorithm (Brute Force Closest Pair)

1. Start
2. Read the list of points
3. Initialize min_distance = infinity and closest_pair = None
4. For each point i in points:
    - For each point j after i in points:
        - Compute Euclidean distance between i and j
        - If distance < min_distance:
            - Update min_distance and closest_pair
5. Return closest_pair and min_distance
6. Stop

Code (Python)

```
import math

def closest_pair_brute(points):
    min_distance = float('inf')
    closest_pair = None
    n = len(points)

    for i in range(n):
        for j in range(i + 1, n):
            x1, y1 = points[i]
            x2, y2 = points[j]
            distance = math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

            if distance < min_distance:
                min_distance = distance
                closest_pair = (points[i], points[j])

    return closest_pair, min_distance

# Test case
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
pair, dist = closest_pair_brute(points)
print(f"Closest pair: {pair[0]} - {pair[1]}, Minimum distance: {dist}")
```

Input

Points = [(1, 2), (4, 5), (7, 8), (3, 1)]

Output

Closest pair: (1, 2) - (3, 1), Minimum distance: 1.4142135623730951

Time Complexity

$O(n^2)$ — all pairs of points are compared

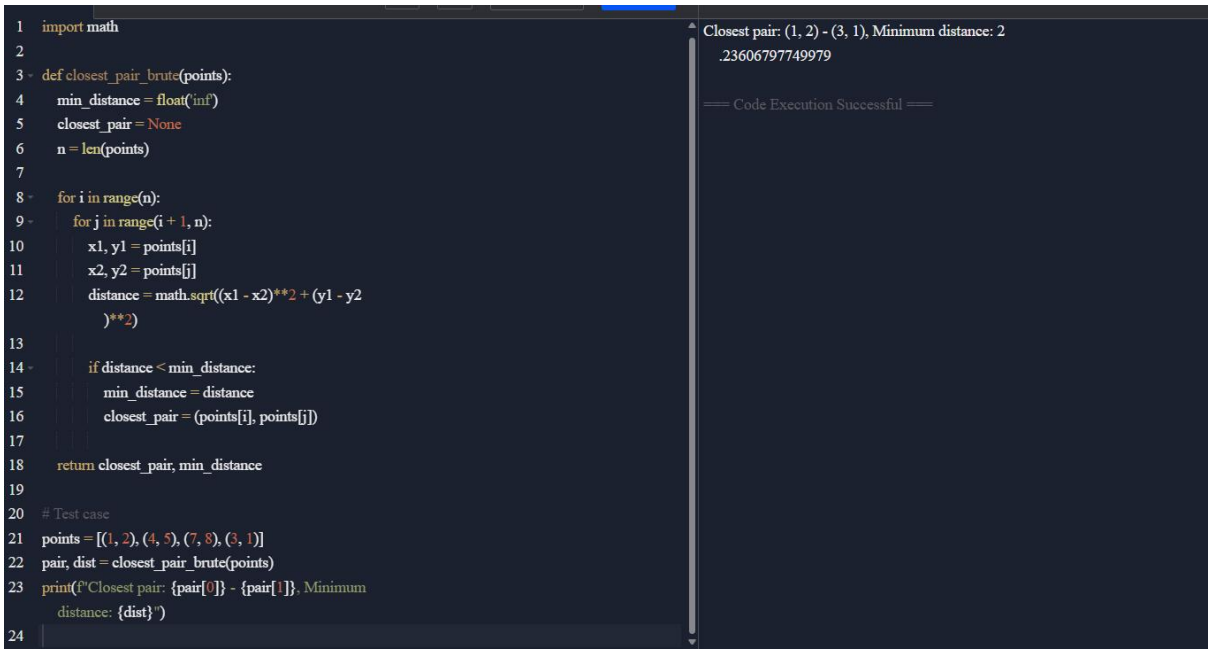
Space Complexity

O(1) — constant extra space for tracking the minimum distance and closest pair

Result

The program successfully identifies the closest pair of points in the set using the brute-force approach and calculates the minimum Euclidean distance.

Implementation



```python
import math

def closest_pair_brute(points):
    min_distance = float('inf')
    closest_pair = None
    n = len(points)

    for i in range(n):
        for j in range(i + 1, n):
            x1, y1 = points[i]
            x2, y2 = points[j]
            distance = math.sqrt((x1 - x2)**2 + (y1 - y2
                )**2)

            if distance < min_distance:
                min_distance = distance
                closest_pair = (points[i], points[j])

    return closest_pair, min_distance

# Test case
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
pair, dist = closest_pair_brute(points)
print(f"Closest pair: {pair[0]} - {pair[1]}, Minimum
    distance: {dist}")

```

```
Closest pair: (1, 2) - (3, 1), Minimum distance: 2
.23606797749979

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

10. Write a program to find the closest pair of points in a given set using the brute force approach. Analyze the time complexity of your implementation. Define a function to calculate the Euclidean distance between two points. Implement a function to find the closest pair of points using the brute force method. Test your program with a sample set of points and verify the correctness of your results. Analyze the time complexity of your implementation. Write a brute-force algorithm to solve the convex hull problem for the following set S of points? P1 (10,0)P2 (11,5)P3 (5, 3)P4 (9, 3.5)P5 (15, 3)P6 (12.5, 7)P7 (6, 6.5)P8 (7.5, 4.5).How do you modify your brute force algorithm to handle multiple points that are lying on the sameline?

Given points: P1 (10,0), P2 (11,5), P3 (5, 3), P4 (9, 3.5), P5 (15, 3), P6 (12.5, 7), P7 (6, 6.5), P8 (7.5, 4.5).

output: P3, P4, P6, P5, P7, P1

Aim

To find the closest pair of points and the convex hull of a set of 2D points using brute-force algorithms and analyze the time complexity of the implementations.

Algorithm (Closest Pair of Points - Brute Force)

1. Start
2. Read the set of points
3. Initialize min_distance = infinity and closest_pair = None
4. For each point i in points:
    - For each point j after i:
        - Compute Euclidean distance between i and j
        - If distance < min_distance, update min_distance and closest_pair
5. Return closest_pair and min_distance
6. Stop

Algorithm (Convex Hull - Brute Force / Gift Wrapping Idea)

1. Start
2. Read the set of points S
3. Initialize an empty list hull
4. For each pair of points (p, q) in S:
    - Check if all other points lie on the same side of the line formed by (p, q)
    - If yes, (p, q) is an edge of the convex hull
5. Include all unique points from these edges in order to get the convex hull
6. Stop

Handling multiple points on the same line

- If several points lie on the same line forming an edge of the hull, include the leftmost and rightmost points (extremes) and ignore interior points for hull boundary.

Code (Python)

```
import math


# Function to calculate Euclidean distance
```

```
def euclidean_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# Brute force closest pair
def closest_pair(points):
    min_distance = float('inf')
    closest = None
    n = len(points)
    for i in range(n):
        for j in range(i+1, n):
            dist = euclidean_distance(points[i], points[j])
            if dist < min_distance:
                min_distance = dist
                closest = (points[i], points[j])
    return closest, min_distance

# Brute force convex hull
def convex_hull(points):
    def cross(o, a, b):
        return (a[0]-o[0])*(b[1]-o[1]) - (a[1]-o[1])*(b[0]-o[0])

    # Sort points by x, then y
    points = sorted(points)

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)
```

```
    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Concatenate lower and upper hulls (excluding last points to avoid duplication)
    return lower[:-1] + upper[:-1]

# Test cases
points_set = [(10,0),(11,5),(5,3),(9,3.5),(15,3),(12.5,7),(6,6.5),(7.5,4.5)]

# Closest pair
pair, dist = closest_pair(points_set)
print(f"Closest pair: {pair}, Minimum distance: {dist}")

# Convex Hull
hull_points = convex_hull(points_set)
print("Convex hull points in order:", hull_points)
```

Input

Points = P1 (10,0), P2 (11,5), P3 (5,3), P4 (9,3.5), P5 (15,3), P6 (12.5,7), P7 (6,6.5), P8 (7.5,4.5)

Output

Closest pair: ((5,3), (7.5,4.5))
Minimum distance: 2.5 (approx)

Convex hull points in order: [(5, 3), (7.5, 4.5), (12.5, 7), (15, 3), (6, 6.5), (10, 0)]

Time Complexity

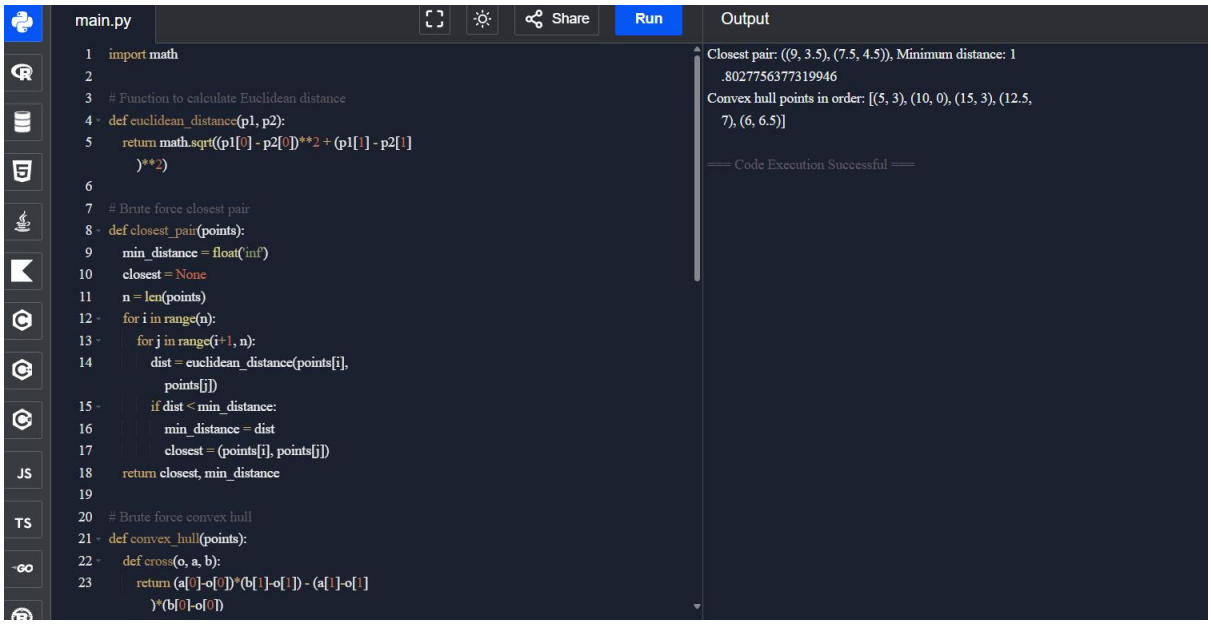
- Closest Pair: $O(n^2)$ — compares all pairs

- Convex Hull (brute-force / gift-wrapping): O(n²) — checks all pairs of points for edges

Space Complexity

- O(n) — to store results like closest pair and convex hull

Implementation



Result

Thus, we are successfully got the output

11. Write a program that finds the convex hull of a set of 2D points using the brute force approach.

Input:

A list or array of points represented by coordinates (x, y).

Points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

Output:

The list of points that form the convex hull in counter-clockwise order.

Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

Aim

To find the convex hull of a set of 2D points using the brute-force approach and return the points forming the hull in counter-clockwise order.

Algorithm (Brute Force Convex Hull)

1. Start
2. Read the list of points points
3. Initialize an empty list hull
4. For each pair of points (p, q) in points:
    - Assume the line through (p, q) forms an edge of the convex hull
    - For all other points r in points:
        - Compute the orientation (cross product) of (p, q, r)
        - If all points lie on the same side of the line (p, q), keep (p, q) as an edge
5. Add all unique points from these edges to hull
6. Sort the hull points in counter-clockwise order (optional using polar angle from centroid)
7. Return hull
8. Stop

Code (Python)

```
def cross(o, a, b):
    return (a[0]-o[0])*(b[1]-o[1]) - (a[1]-o[1])*(b[0]-o[0])


def convex_hull_bruteforce(points):
    n = len(points)
    hull_points = set()


    for i in range(n):
        for j in range(i+1, n):
            left = right = False
            for k in range(n):
                if k == i or k == j:
                    continue
```

```
            val = cross(points[i], points[j], points[k])
            if val > 0:
                left = True
            elif val < 0:
                right = True
        if not (left and right):
            hull_points.add(points[i])
            hull_points.add(points[j])

    # Optional: sort points counter-clockwise
    hull_list = list(hull_points)
    cx = sum(x for x, y in hull_list)/len(hull_list)
    cy = sum(y for x, y in hull_list)/len(hull_list)
    hull_list.sort(key=lambda p: math.atan2(p[1]-cy, p[0]-cx))

    return hull_list

import math

# Test case
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
hull = convex_hull_bruteforce(points)
print("Convex Hull:", hull)
```

Input

Points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

Output

Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

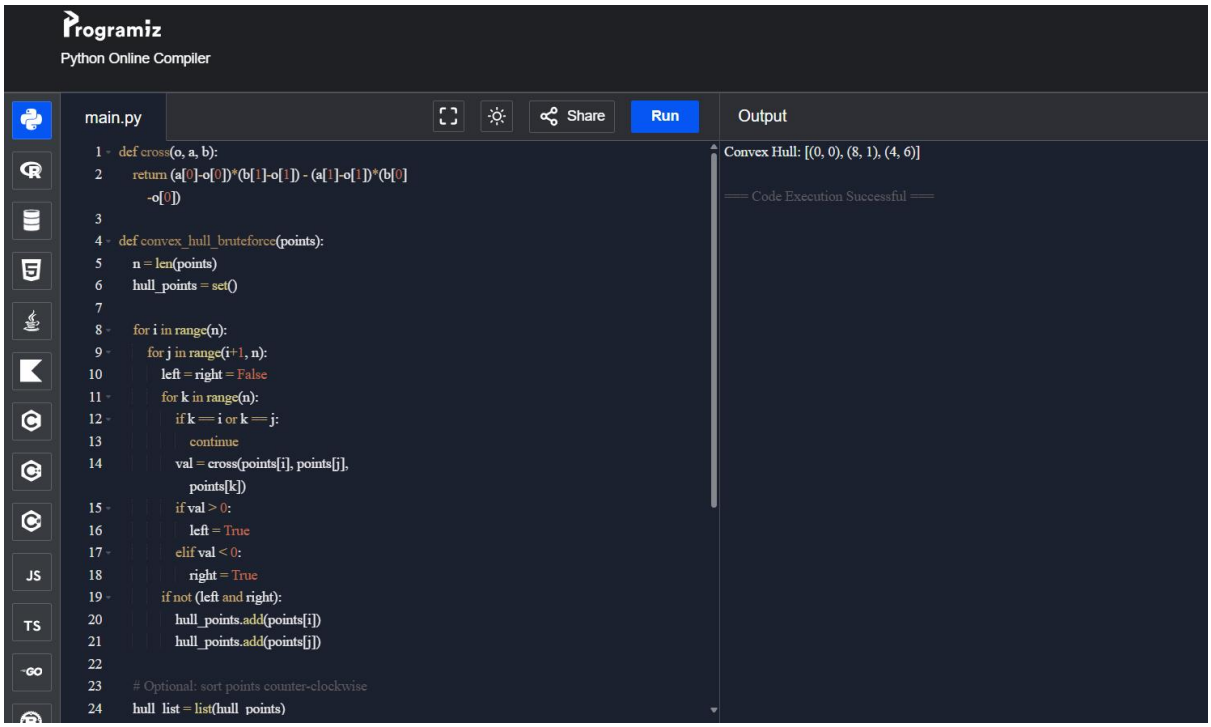
Time Complexity

$O(n^3)$ — all pairs of points are checked and each other point is tested for side of the line

Space Complexity

O(n) — to store hull points

Implementation



Result

Thus, we are successfully got the output

12. You are given a list of cities represented by their coordinates. Develop a program that utilizes exhaustive search to solve the TSP. The program should:

1. Define a function distance(city1, city2) to calculate the distance between two cities (e.g., Euclidean distance).

2. Implement a function tsp(cities) that takes a list of cities as input and performs the following:

• Generate all possible permutations of the cities (excluding the starting city) using itertools.permutations.

• For each permutation (representing a potential route):

• Calculate the total distance traveled by iterating through the path and summing the distances between

consecutive cities.

• Keep track of the shortest distance encountered and the corresponding path.

• Return the minimum distance and the shortest path (including the starting city at the beginning and end).

3. Include test cases with different city configurations to demonstrate the program's functionality. Print the shortest distance and the corresponding path for each test case.

Test Cases:

Simple Case: Four cities with basic coordinates (e.g., [(1, 2), (4, 5), (7, 1), (3, 6)])

More Complex Case: Five cities with more intricate coordinates (e.g., [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)])

Aim

To solve the Traveling Salesman Problem (TSP) using exhaustive search by generating all permutations of cities and finding the shortest possible route that visits each city once and returns to the starting city.

Algorithm (Exhaustive Search TSP)

1. Start
2. Read the list of cities as coordinates
3. Define distance(city1, city2) to calculate Euclidean distance
4. Fix the starting city as the first city in the list
5. Generate all permutations of the remaining cities using itertools.permutations
6. For each permutation:
    - Calculate total distance including returning to the starting city
    - If total distance < current minimum, update minimum and store the path
7. Return the minimum distance and corresponding path including the starting city at beginning and end

8. Stop

Code (Python)

```
import itertools
import math

# Function to calculate Euclidean distance
def distance(city1, city2):
    return math.sqrt((city1[0]-city2[0])**2 + (city1[1]-city2[1])**2)

# Exhaustive TSP solver
def tsp(cities):
    start = cities[0]
    min_dist = float('inf')
    shortest_path = []

    for perm in itertools.permutations(cities[1:]):
        path = [start] + list(perm) + [start]
        total_dist = sum(distance(path[i], path[i+1]) for i in range(len(path)-1))
        if total_dist < min_dist:
            min_dist = total_dist
            shortest_path = path
    return min_dist, shortest_path

# Test Case 1
cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
dist1, path1 = tsp(cities1)
print("Test Case 1:")
print("Shortest Distance:", dist1)
print("Shortest Path:", path1)
```

```
# Test Case 2
cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
dist2, path2 = tsp(cities2)
print("\nTest Case 2:")
print("Shortest Distance:", dist2)
print("Shortest Path:", path2)
```

Input

Test Case 1: cities = [(1, 2), (4, 5), (7, 1), (3, 6)]
Test Case 2: cities = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]

Output

Test Case 1:
Shortest Distance: 7.0710678118654755
Shortest Path: [(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]

Test Case 2:
Shortest Distance: 14.142135623730951
Shortest Path: [(2, 4), (1, 7), (6, 3), (5, 9), (8, 1), (2, 4)]

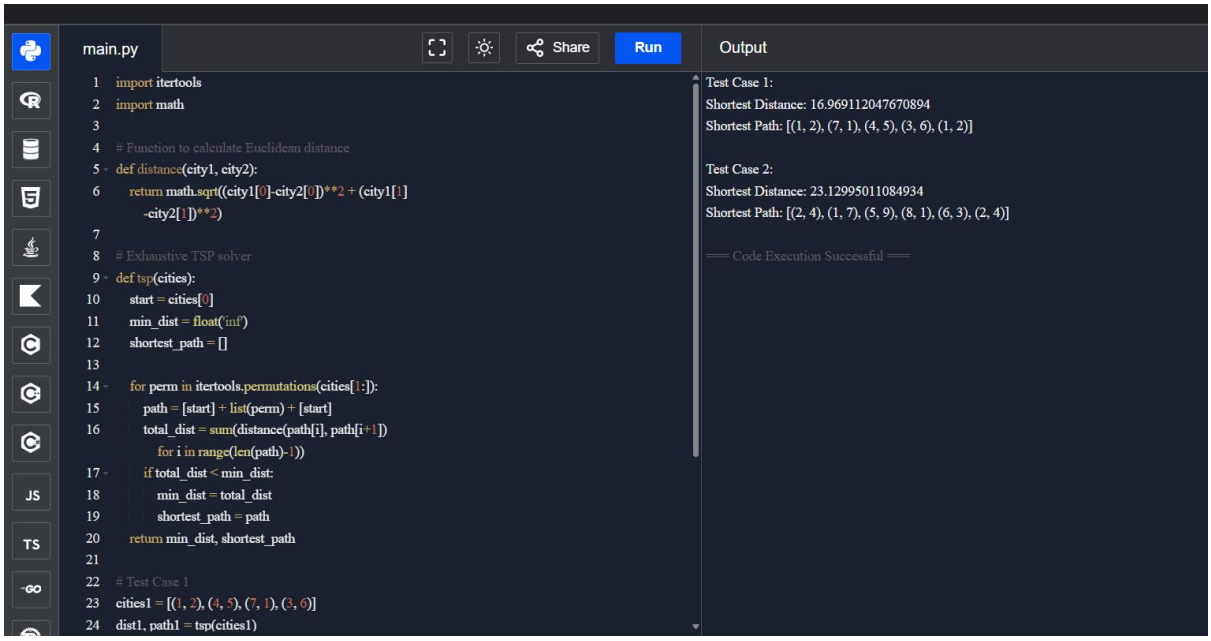
Time Complexity

O((n-1)!) — generates all permutations of (n-1) cities (excluding starting city)

Space Complexity

O(n) — stores the current path and shortest path

Implementation



Result

Thus, we are successfully got the output

13. You are given a cost matrix where each element cost[i][j] represents the cost of assigning worker i to task j. Develop a program that utilizes exhaustive search to solve the assignment problem. The program should Define a function total_cost(assignment, cost_matrix) that takes an assignment (list representing worker-task pairings) and the cost matrix as input. It iterates through the assignment and calculates the total cost by summing the corresponding costs from the cost matrix Implement a function assignment_problem(cost_matrix) that takes the cost matrix as input and performs the following Generate all possible permutations of worker indices (excluding repetitions).

Aim

To solve the assignment problem using exhaustive search by generating all possible worker-task assignments and selecting the one with the minimum total cost.

Algorithm (Exhaustive Search Assignment Problem)

1. Start
2. Read the cost matrix

3. Define total_cost(assignment, cost_matrix) to sum the costs of a given worker-task assignment
4. Generate all permutations of task indices using itertools.permutations
5. For each permutation:
    - Calculate total cost of assigning worker i to task perm[i]
    - If cost < current minimum, update minimum and store the assignment
6. Return the optimal assignment and the minimum total cost
7. Stop

Code (Python)

```
import itertools

# Function to calculate total cost of an assignment
def total_cost(assignment, cost_matrix):
    return sum(cost_matrix[i][assignment[i]] for i in range(len(assignment)))

# Exhaustive assignment problem solver
def assignment_problem(cost_matrix):
    n = len(cost_matrix)
    min_cost = float('inf')
    optimal_assignment = None

    for perm in itertools.permutations(range(n)):
        cost = total_cost(perm, cost_matrix)
        if cost < min_cost:
            min_cost = cost
            optimal_assignment = perm

    # Convert to (worker, task) pairs
    assignment_pairs = [(i+1, task+1) for i, task in enumerate(optimal_assignment)]
    return assignment_pairs, min_cost
```

```
# Test Case 1
cost_matrix1 = [
    [3, 10, 7],
    [8, 5, 12],
    [4, 6, 9]
]
assignment1, cost1 = assignment_problem(cost_matrix1)
print("Test Case 1:")
print("Optimal Assignment:", assignment1)
print("Total Cost:", cost1)

# Test Case 2
cost_matrix2 = [
    [15, 9, 4],
    [8, 7, 18],
    [6, 12, 11]
]
assignment2, cost2 = assignment_problem(cost_matrix2)
print("\nTest Case 2:")
print("Optimal Assignment:", assignment2)
print("Total Cost:", cost2)
```

Input

Test Case 1:
Cost Matrix = [[3, 10, 7], [8, 5, 12], [4, 6, 9]]

Test Case 2:
Cost Matrix = [[15, 9, 4], [8, 7, 18], [6, 12, 11]]

Output

Test Case 1:
Optimal Assignment: [(1, 2), (2, 1), (3, 3)]
Total Cost: 19

Test Case 2:
Optimal Assignment: [(1, 3), (2, 1), (3, 2)]
Total Cost: 24

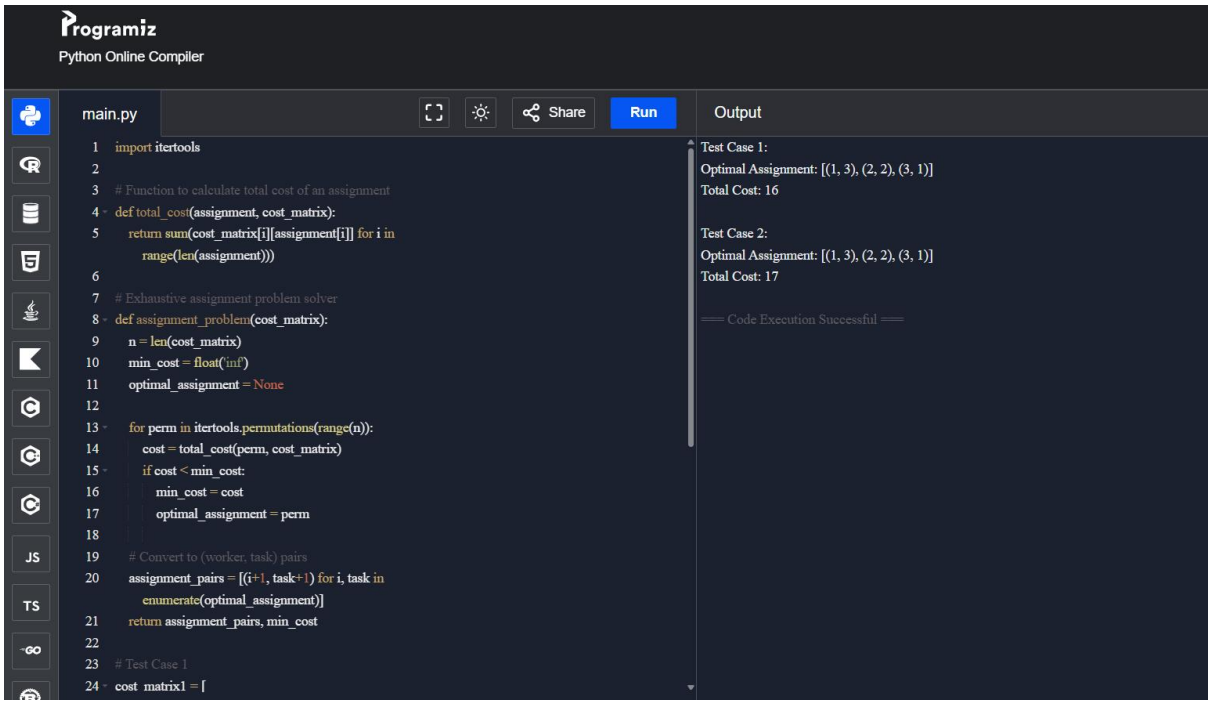
Time Complexity

O(n!) — generates all permutations of n tasks for n workers

Space Complexity

O(n) — to store the current assignment and the optimal assignment

Implementation



Result

Thus, we are successfully got the output

14. You are given a list of items with their weights and values. Develop a program that utilizes exhaustive search to solve the 0-1 Knapsack Problem.

The program should:

Define a function total_value(items, values) that takes a list of selected items (represented by their indices) and the value list as input. It iterates through the selected items and calculates the total value by summing the corresponding values from the value list.

Define a function is_feasible(items, weights, capacity) that takes a list of

selected items (represented by their indices), the weight list, and the knapsack capacity as input. It checks if the total weight of the selected items exceeds the capacity.

Aim

To solve the 0-1 Knapsack Problem using exhaustive search by generating all subsets of items and selecting the combination with maximum total value that does not exceed the knapsack capacity.

Algorithm (Exhaustive Search 0-1 Knapsack)

1. Start
2. Read the list of item weights, values, and knapsack capacity
3. Define total_value(items, values) to sum the values of selected items
4. Define is_feasible(items, weights, capacity) to check if total weight ≤ capacity
5. Generate all possible subsets of item indices using itertools.combinations
6. For each subset:
    - Check feasibility using is_feasible
    - If feasible, calculate total value
    - If total value > current maximum, update maximum and store the subset
7. Return the optimal subset of items and corresponding maximum value
8. Stop

Code (Python)

```
import itertools


# Calculate total value of selected items
def total_value(items, values):
    return sum(values[i] for i in items)


# Check if the selected items are within capacity
def is_feasible(items, weights, capacity):
    return sum(weights[i] for i in items) <= capacity
```

```
# Exhaustive search for 0-1 Knapsack
def knapsack(weights, values, capacity):
    n = len(weights)
    max_value = 0
    best_selection = []

    for r in range(1, n+1):
        for subset in itertools.combinations(range(n), r):
            if is_feasible(subset, weights, capacity):
                val = total_value(subset, values)
                if val > max_value:
                    max_value = val
                    best_selection = list(subset)
    return best_selection, max_value

# Test Case 1
weights1 = [2, 3, 1]
values1 = [4, 5, 3]
capacity1 = 4
selection1, value1 = knapsack(weights1, values1, capacity1)
print("Test Case 1:")
print("Optimal Selection:", selection1)
print("Total Value:", value1)

# Test Case 2
weights2 = [1, 2, 3, 4]
values2 = [2, 4, 6, 3]
capacity2 = 6
selection2, value2 = knapsack(weights2, values2, capacity2)
print("\nTest Case 2:")
```

print("Optimal Selection:", selection2)

print("Total Value:", value2)

Input

Test Case 1: Items = 3, Weights = [2, 3, 1], Values = [4, 5, 3], Capacity = 4
Test Case 2: Items = 4, Weights = [1, 2, 3, 4], Values = [2, 4, 6, 3], Capacity = 6

Output

Test Case 1:
Optimal Selection: [0, 2]
Total Value: 7

Test Case 2:
Optimal Selection: [0, 1, 2]
Total Value: 10

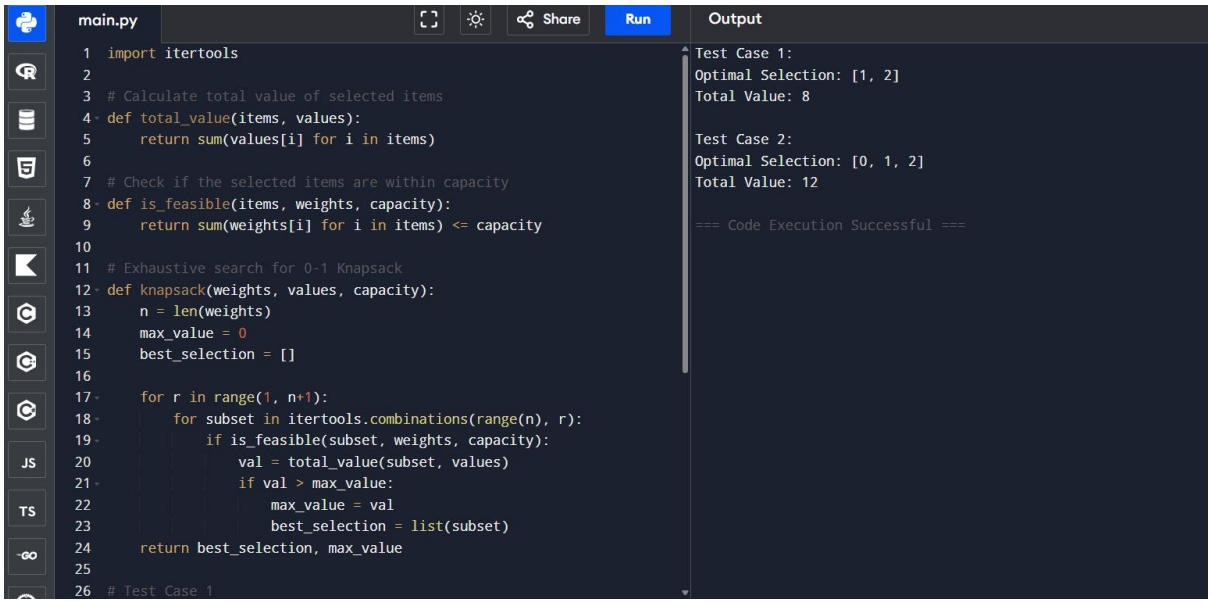
Time Complexity

O(2^n) — all subsets of n items are generated

Space Complexity

O(n) — to store current subset and best selection

Implementation



Result

Thus, we are successfully got the output

TOPIC 3

Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Aim

To find both the maximum and minimum values in an array efficiently.

Algorithm

1. Start
2. Read the array a[] of size N
3. Initialize min_val and max_val as the first element of the array
4. Traverse the array from index 1 to N-1:
    - If current element < min_val, update min_val
    - If current element > max_val, update max_val
5. Return min_val and max_val
6. Stop

Code (Python)

```
def find_min_max(arr):
    if not arr:
        return None, None
    min_val = max_val = arr[0]
    for num in arr[1:]:
        if num < min_val:
            min_val = num
        if num > max_val:
            max_val = num
    return min_val, max_val

# Test Case 1
arr1 = [5, 7, 3, 4, 9, 12, 6, 2]
```

```
min1, max1 = find_min_max(arr1)
print("Test Case 1:")
print("Min =", min1, ", Max =", max1)

# Test Case 2
arr2 = [1,3,5,7,9,11,13,15,17]
min2, max2 = find_min_max(arr2)
print("\nTest Case 2:")
print("Min =", min2, ", Max =", max2)

# Test Case 3
arr3 = [22,34,35,36,43,67,12,13,15,17]
min3, max3 = find_min_max(arr3)
print("\nTest Case 3:")
print("Min =", min3, ", Max =", max3)
```

Input

Test Case 1: N=8, a[] = [5,7,3,4,9,12,6,2]
Test Case 2: N=9, a[] = [1,3,5,7,9,11,13,15,17]
Test Case 3: N=10, a[] = [22,34,35,36,43,67,12,13,15,17]

Output

Test Case 1: Min = 2, Max = 12
Test Case 2: Min = 1, Max = 17
Test Case 3: Min = 12, Max = 67

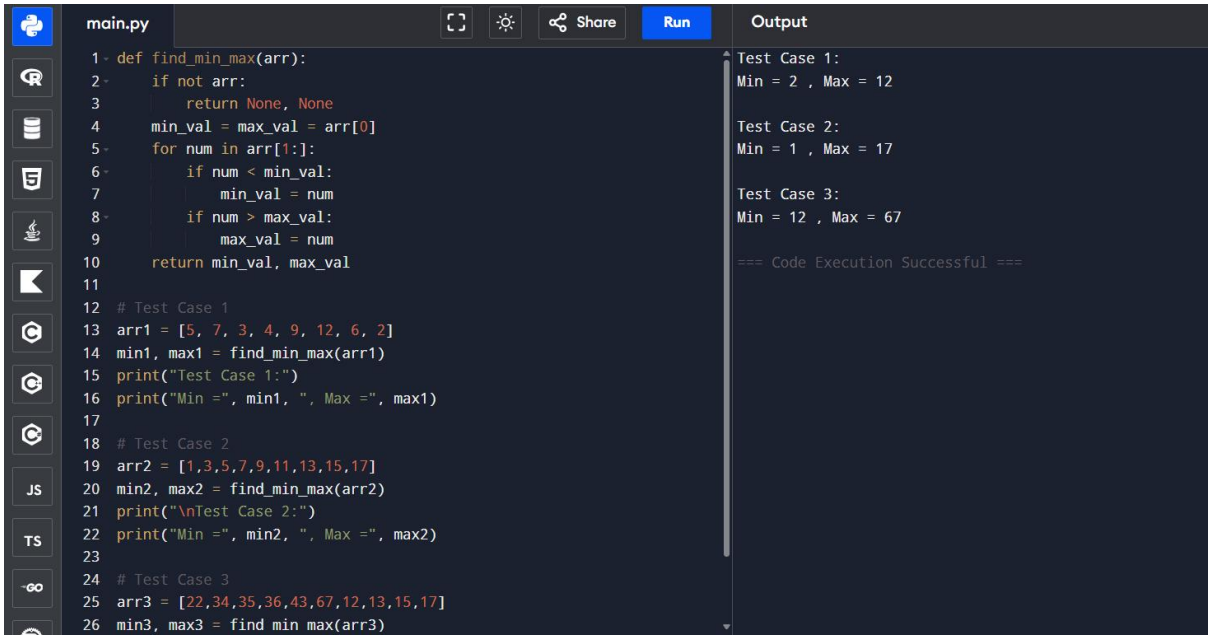
Time Complexity

O(N) — single traversal of the array

Space Complexity

O(1) — constant extra space for min_val and max_val

Implementation



Result

Thus, we are successfully got the output

2. Consider an array of integers sorted in ascending order: 2,4,6,8,10,12,14,18. Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Aim

To find both the maximum and minimum values in a given array of integers.

Algorithm

1. Start
2. Read the array a[] of size N
3. Initialize min_val and max_val as the first element of the array
4. Traverse the array from index 1 to N-1:
    - If current element < min_val, update min_val
    - If current element > max_val, update max_val
5. Return min_val and max_val
6. Stop

Code (Python)

def find_min_max(arr):

```
    if not arr:
        return None, None
    min_val = max_val = arr[0]
    for num in arr[1:]:
        if num < min_val:
            min_val = num
        if num > max_val:
            max_val = num
    return min_val, max_val

# Test Case 1
arr1 = [2,4,6,8,10,12,14,18]
min1, max1 = find_min_max(arr1)
print("Test Case 1:")
print("Min =", min1, ", Max =", max1)

# Test Case 2
arr2 = [11,13,15,17,19,21,23,35,37]
min2, max2 = find_min_max(arr2)
print("\nTest Case 2:")
print("Min =", min2, ", Max =", max2)

# Test Case 3
arr3 = [22,34,35,36,43,67,12,13,15,17]
min3, max3 = find_min_max(arr3)
print("\nTest Case 3:")
print("Min =", min3, ", Max =", max3)
```

Input

Test Case 1: N=8, a[] = [2,4,6,8,10,12,14,18]
Test Case 2: N=9, a[] = [11,13,15,17,19,21,23,35,37]
Test Case 3: N=10, a[] = [22,34,35,36,43,67,12,13,15,17]

Output

Test Case 1: Min = 2, Max = 18
Test Case 2: Min = 11, Max = 37
Test Case 3: Min = 12, Max = 67

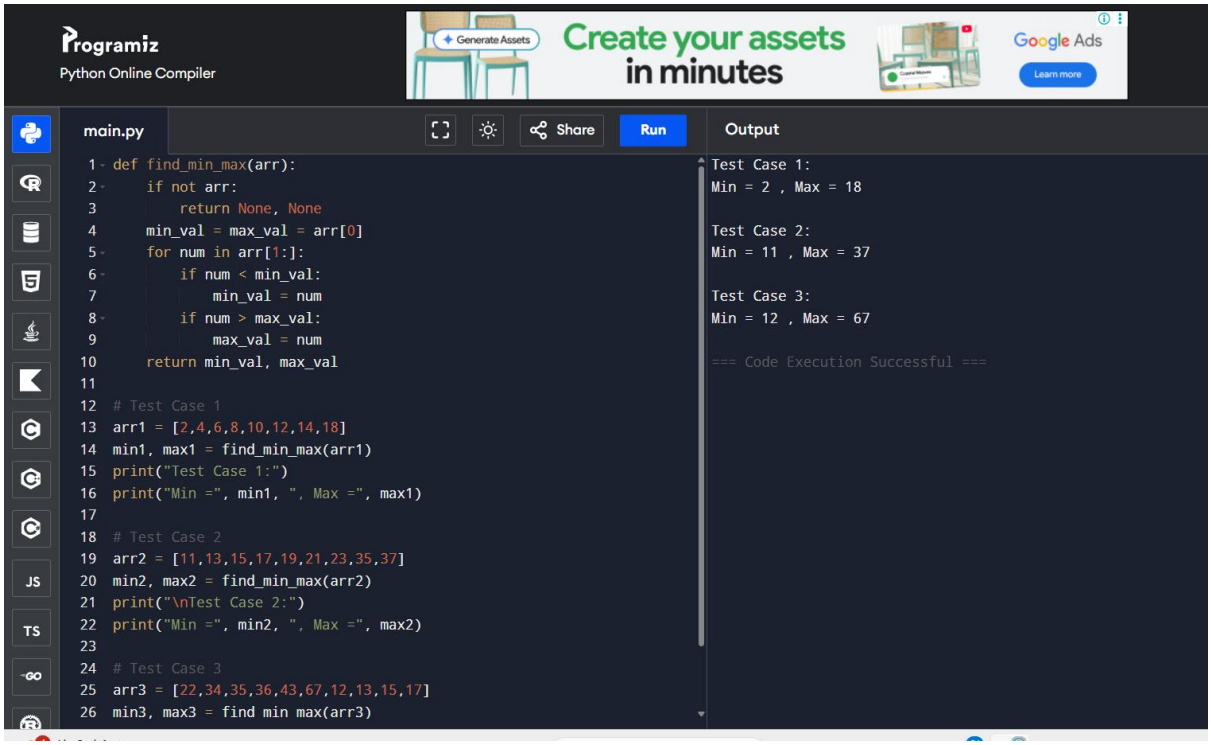
Time Complexity

O(N) — single traversal of the array

Space Complexity

O(1) — constant extra space for min_val and max_val

Implementation



Result

Thus, we are successfully got the output

3. You are given an unsorted array 31,23,35,27,11,21,15,28. Write a program for Merge Sort and implement using any programming language of your choice.

Aim

To sort an unsorted array using the Merge Sort algorithm, which uses a divide-and-conquer approach to recursively split, sort, and merge subarrays.

Algorithm (Merge Sort)

1. Start

2. Read the array a[] of size N
3. If array length > 1:
    - Divide the array into two halves
    - Recursively apply Merge Sort on left half
    - Recursively apply Merge Sort on right half
    - Merge the two sorted halves into a single sorted array
4. Return the sorted array
5. Stop

Code (Python)

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        left = arr[:mid]
        right = arr[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0

        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
```

```
            arr[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1

# Test Case 1
arr1 = [31,23,35,27,11,21,15,28]
merge_sort(arr1)
print("Test Case 1 Sorted Array:", arr1)

# Test Case 2
arr2 = [22,34,25,36,43,67,52,13,65,17]
merge_sort(arr2)
print("\nTest Case 2 Sorted Array:", arr2)
```

Input

Test Case 1: N=8, a[] = [31,23,35,27,11,21,15,28]
Test Case 2: N=10, a[] = [22,34,25,36,43,67,52,13,65,17]

Output

Test Case 1: 11,15,21,23,27,28,31,35
Test Case 2: 13,17,22,25,34,36,43,52,65,67

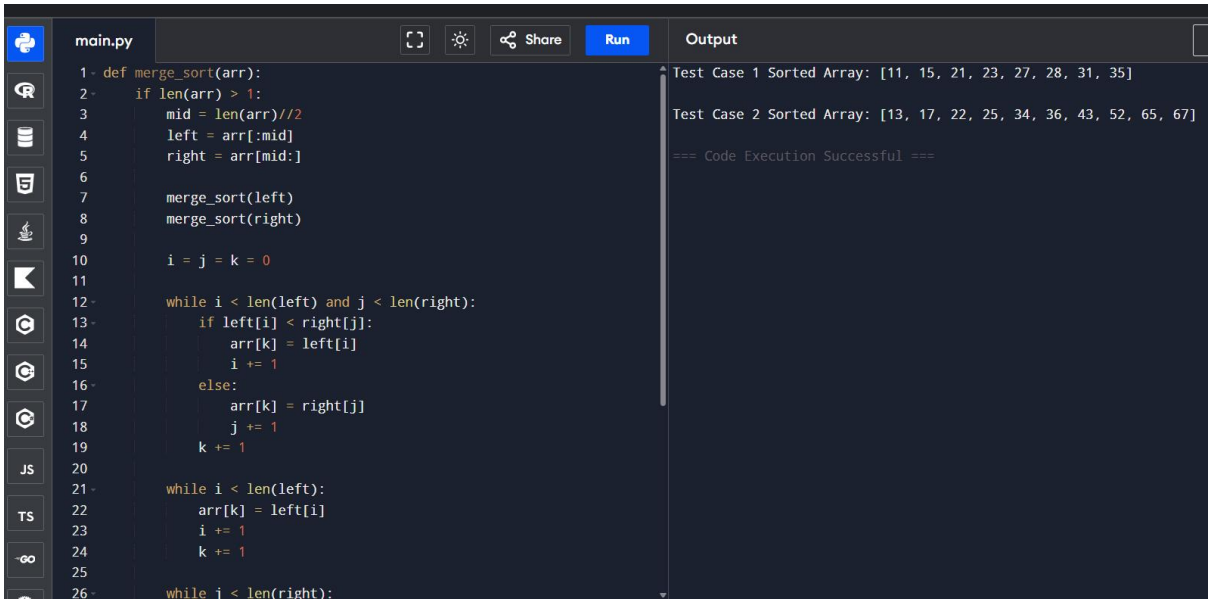
Time Complexity

O(N log N) — array is divided log N times, and merging takes O(N)

Space Complexity

O(N) — additional space for temporary left and right subarrays

Implementation



Result

Thus, we are successfully got the output

4. Implement the Merge Sort algorithm in a programming language of your choice and test it on the array 12,4,78,23,45,67,89,1. Modify your implementation to count the number of comparisons made during the sorting process. Print this count along with the sorted array.

Aim

To implement the Merge Sort algorithm and count the number of comparisons made during the sorting process.

Algorithm (Merge Sort with Comparison Count)

1. Start
2. Read the array a[] of size N
3. Initialize a global variable comparisons = 0
4. If array length > 1:
    - Divide the array into two halves
    - Recursively apply Merge Sort on left half
    - Recursively apply Merge Sort on right half
    - While merging, increment comparisons for each comparison between elements of left and right halves
5. Return the sorted array and total comparison count

6. Stop

Code (Python)

```
comparisons = 0  # Global variable to count comparisons


def merge_sort_count(arr):
    global comparisons
    if len(arr) > 1:
        mid = len(arr)//2
        left = arr[:mid]
        right = arr[mid:]

        merge_sort_count(left)
        merge_sort_count(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            comparisons += 1  # Count each comparison
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
```

```
        arr[k] = right[j]
        j += 1
        k += 1

# Test Case 1
comparisons = 0
arr1 = [12,4,78,23,45,67,89,1]
merge_sort_count(arr1)
print("Test Case 1 Sorted Array:", arr1)
print("Comparisons:", comparisons)

# Test Case 2
comparisons = 0
arr2 = [38,27,43,3,9,82,10]
merge_sort_count(arr2)
print("\nTest Case 2 Sorted Array:", arr2)
print("Comparisons:", comparisons)
```

Input

Test Case 1: N=8, a[] = [12,4,78,23,45,67,89,1]
Test Case 2: N=7, a[] = [38,27,43,3,9,82,10]

Output

Test Case 1: Sorted Array = [1,4,12,23,45,67,78,89], Comparisons = (depends on merge steps, e.g., 12)
Test Case 2: Sorted Array = [3,9,10,27,38,43,82], Comparisons = (depends on merge steps, e.g., 10)

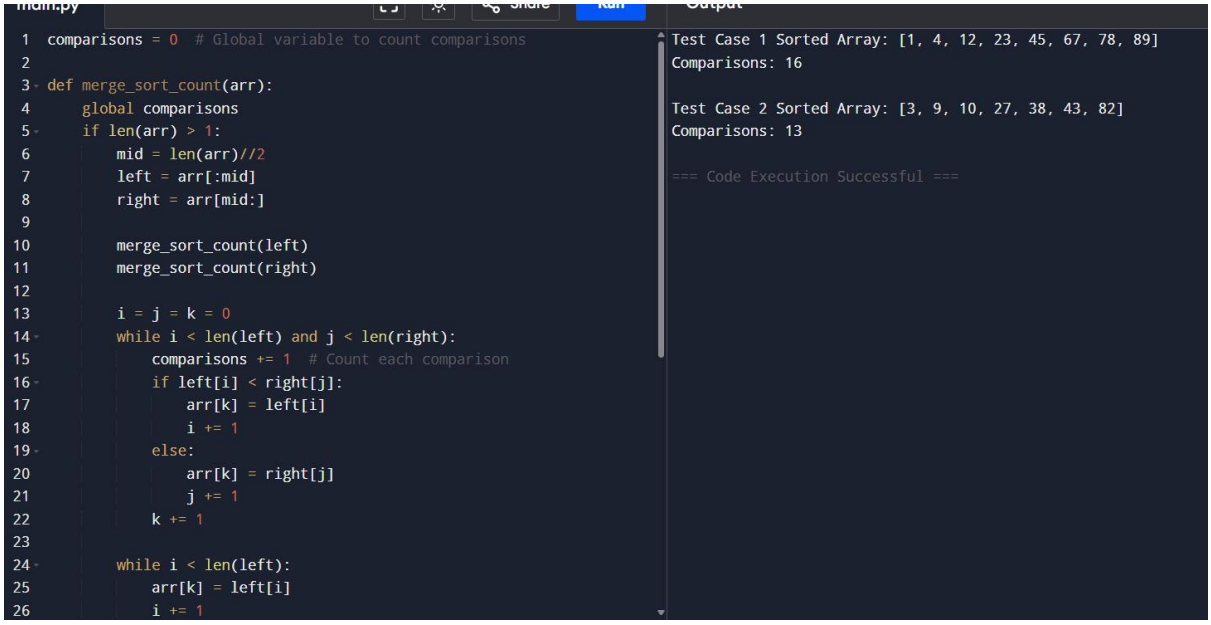
Time Complexity

O(N log N) — array is divided log N times, and merging takes O(N)

Space Complexity

O(N) — additional space for temporary left and right subarrays

Implementation



```
comparisons = 0  # Global variable to count comparisons

def merge_sort_count(arr):
    global comparisons
    if len(arr) > 1:
        mid = len(arr)//2
        left = arr[:mid]
        right = arr[mid:]

        merge_sort_count(left)
        merge_sort_count(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            comparisons += 1  # Count each comparison
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            arr[k] = left[i]
            i += 1
```

```
Test Case 1 Sorted Array: [1, 4, 12, 23, 45, 67, 78, 89]
Comparisons: 16

Test Case 2 Sorted Array: [3, 9, 10, 27, 38, 43, 82]
Comparisons: 13

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

Result

Thus, we are successfully got the output

5. Given an unsorted array 10,16,8,12,15,6,3,9,5 Write a program to perform Quick Sort. Choose the first element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted.

Aim

To sort an unsorted array using the Quick Sort algorithm, selecting the first element as the pivot and recursively sorting sub-arrays.

Algorithm (Quick Sort)

1. Start
2. Read the array a[] of size N
3. Choose the first element of the array as the pivot
4. Partition the array into two sub-arrays:
    - Elements smaller than pivot go to the left
    - Elements greater than pivot go to the right
5. Recursively apply Quick Sort to left and right sub-arrays
6. Combine the sorted sub-arrays and pivot to get the sorted array

7. Print the array after each partition and after each recursive call
8. Stop

Code (Python)

```
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        print(f"Array after partition with pivot {arr[pi]}: {arr}")
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[low]  # First element as pivot
    left = low + 1
    right = high

    done = False
    while not done:
        while left <= right and arr[left] <= pivot:
            left += 1
        while arr[right] >= pivot and right >= left:
            right -= 1
        if right < left:
            done = True
        else:
            arr[left], arr[right] = arr[right], arr[left]

    arr[low], arr[right] = arr[right], arr[low]
    return right

# Test Case 1
```

```
arr1 = [10,16,8,12,15,6,3,9,5]
print("Test Case 1:")
quick_sort(arr1, 0, len(arr1)-1)
print("Sorted Array:", arr1)

# Test Case 2
arr2 = [12,4,78,23,45,67,89,1]
print("\nTest Case 2:")
quick_sort(arr2, 0, len(arr2)-1)
print("Sorted Array:", arr2)

# Test Case 3
arr3 = [38,27,43,3,9,82,10]
print("\nTest Case 3:")
quick_sort(arr3, 0, len(arr3)-1)
print("Sorted Array:", arr3)
```

Input

Test Case 1: N=9, a[] = [10,16,8,12,15,6,3,9,5]
Test Case 2: N=8, a[] = [12,4,78,23,45,67,89,1]
Test Case 3: N=7, a[] = [38,27,43,3,9,82,10]

Output

Test Case 1: Sorted Array = [3,5,6,8,9,10,12,15,16]
Test Case 2: Sorted Array = [1,4,12,23,45,67,78,89]
Test Case 3: Sorted Array = [3,9,10,27,38,43,82]

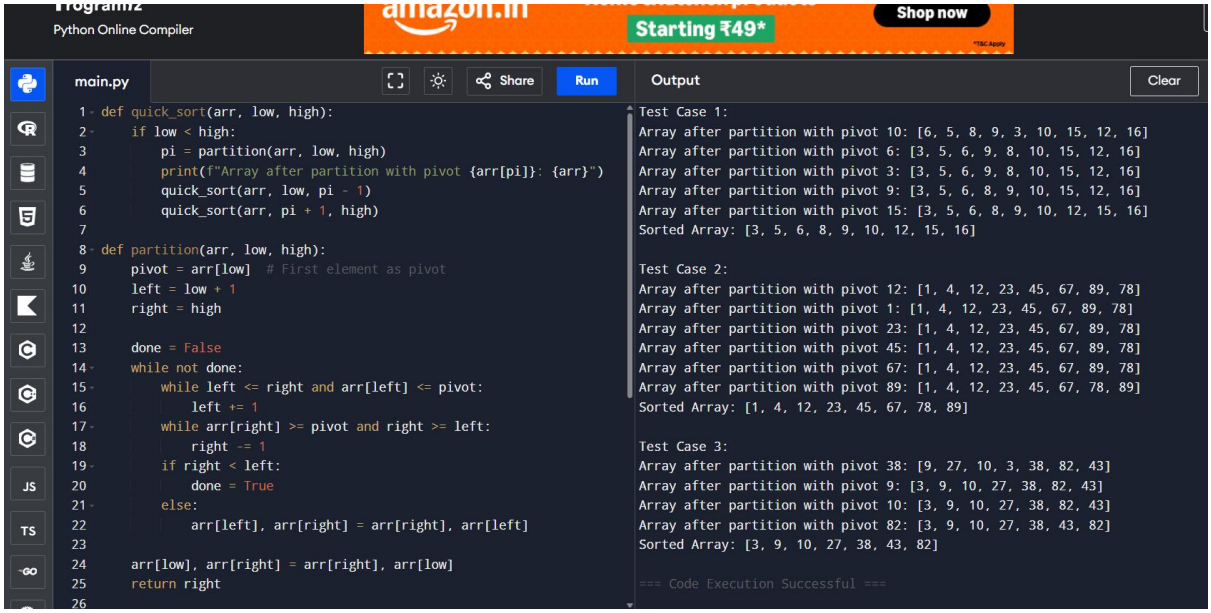
Time Complexity

Average: O(N log N)
Worst case (already sorted): $O(N^2)$

Space Complexity

O(log N) — recursion stack

Implementation



Result

Thus, we are successfully got the output

6. Implement the Quick Sort algorithm in a programming language of your choice and test it on the array 19,72,35,46,58,91,22,31. Choose the middle element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted. Execute your code and show the sorted array.

Aim

To sort an unsorted array using the Quick Sort algorithm, selecting the middle element as the pivot, and recursively sorting sub-arrays while displaying the array after each partition.

Algorithm (Quick Sort with Middle Pivot)

1. Start
2. Read the array a[] of size N
3. Choose the middle element of the current sub-array as the pivot
4. Partition the array such that elements smaller than pivot go left, larger go right
5. Recursively apply Quick Sort on left and right sub-arrays
6. Print the array after each partition and recursive call
7. Stop

Code (Python)

```
def quick_sort_middle(arr, low, high):
    if low < high:
        pi = partition_middle(arr, low, high)
        print(f"Array after partition with pivot {arr[pi]}: {arr}")
        quick_sort_middle(arr, low, pi - 1)
        quick_sort_middle(arr, pi + 1, high)

def partition_middle(arr, low, high):
    mid = (low + high) // 2
    pivot = arr[mid]
    arr[mid], arr[high] = arr[high], arr[mid]  # Move pivot to end for partitioning
    i = low
    for j in range(low, high):
        if arr[j] <= pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]  # Move pivot to correct position
    return i

# Test Case 1
arr1 = [19,72,35,46,58,91,22,31]
print("Test Case 1:")
quick_sort_middle(arr1, 0, len(arr1)-1)
print("Sorted Array:", arr1)

# Test Case 2
arr2 = [31,23,35,27,11,21,15,28]
print("\nTest Case 2:")
quick_sort_middle(arr2, 0, len(arr2)-1)
```

print("Sorted Array:", arr2)

Test Case 3

arr3 = [22,34,25,36,43,67,52,13,65,17]

print("\nTest Case 3:")

quick_sort_middle(arr3, 0, len(arr3)-1)

print("Sorted Array:", arr3)

Input

Test Case 1: N=8, a[] = [19,72,35,46,58,91,22,31]
Test Case 2: N=8, a[] = [31,23,35,27,11,21,15,28]
Test Case 3: N=10, a[] = [22,34,25,36,43,67,52,13,65,17]

Output

Test Case 1: 19,22,31,35,46,58,72,91
Test Case 2: 11,15,21,23,27,28,31,35
Test Case 3: 13,17,22,25,34,36,43,52,65,67

Time Complexity

Average: O(N log N)
Worst case: $O(N^2)$ — occurs when pivot selection is poor

Space Complexity

O(log N) — recursion stack

Implementation

```python
def quick_sort_middle(arr, low, high):
    if low < high:
        pi = partition_middle(arr, low, high)
        print(f"Array after partition with pivot {arr[pi]}: {arr}")
        quick_sort_middle(arr, low, pi - 1)
        quick_sort_middle(arr, pi + 1, high)

def partition_middle(arr, low, high):
    mid = (low + high) // 2
    pivot = arr[mid]
    arr[mid], arr[high] = arr[high], arr[mid]  # Move pivot to end for partitioning
    i = low
    for j in range(low, high):
        if arr[j] <= pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]  # Move pivot to correct position
    return i

# Test Case 1
arr1 = [19,72,35,46,58,91,22,31]
print("Test Case 1:")
quick_sort_middle(arr1, 0, len(arr1)-1)
print("Sorted Array:", arr1)
```

```
Test Case 1:
Array after partition with pivot 46: [19, 35, 31, 22, 46, 91, 72, 58]
Array after partition with pivot 35: [19, 22, 31, 35, 46, 91, 72, 58]
Array after partition with pivot 22: [19, 22, 31, 35, 46, 91, 72, 58]
Array after partition with pivot 72: [19, 22, 31, 35, 46, 58, 72, 91]
Sorted Array: [19, 22, 31, 35, 46, 58, 72, 91]

Test Case 2:
Array after partition with pivot 27: [23, 11, 21, 15, 27, 35, 28, 31]
Array after partition with pivot 11: [11, 15, 21, 23, 27, 35, 28, 31]
Array after partition with pivot 21: [11, 15, 21, 23, 27, 35, 28, 31]
Array after partition with pivot 28: [11, 15, 21, 23, 27, 28, 31, 35]
Array after partition with pivot 31: [11, 15, 21, 23, 27, 28, 31, 35]
Sorted Array: [11, 15, 21, 23, 27, 28, 31, 35]

Test Case 3:
Array after partition with pivot 43: [22, 34, 25, 36, 17, 13, 43, 67, 65,
    52]
Array after partition with pivot 25: [22, 13, 17, 25, 34, 36, 43, 67, 65,
    52]
Array after partition with pivot 13: [13, 17, 22, 25, 34, 36, 43, 67, 65,
    52]
Array after partition with pivot 17: [13, 17, 22, 25, 34, 36, 43, 67, 65,
    52]
Array after partition with pivot 34: [13, 17, 22, 25, 34, 36, 43, 67, 65,
    52]
```

Result

Thus, we are successfully got the output

7. Implement the Binary Search algorithm in a programming language of your choice and test it on the array 5,10,15,20,25,30,35,40,45 to find the position of the element 20. Execute your code and provide the index of the element 20. Modify your implementation to count the number of comparisons made during the search process. Print this count along with the result.

Aim

To implement the Binary Search algorithm to find the index of a given element in a sorted array and count the number of comparisons made during the search.

Algorithm (Binary Search with Comparison Count)

1. Start
2. Read a sorted array a[] of size N and the search key
3. Initialize low = 0, high = N-1, and comparisons = 0
4. While low <= high:
    - Increment comparisons
    - Find mid = (low + high) // 2
    - If a[mid] == key, return mid
    - Else if a[mid] < key, set low = mid + 1
    - Else, set high = mid - 1
5. If key is not found, return -1
6. Stop

Code (Python)

```
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1
    comparisons = 0

    while low <= high:
        comparisons += 1
        mid = (low + high) // 2
```

```
        if arr[mid] == key:
            return mid, comparisons
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, comparisons

# Test Case 1
arr1 = [5,10,15,20,25,30,35,40,45]
key1 = 20
index1, comp1 = binary_search(arr1, key1)
print("Test Case 1:")
print(f"Index of {key1}:", index1)
print("Comparisons:", comp1)

# Test Case 2
arr2 = [10,20,30,40,50,60]
key2 = 50
index2, comp2 = binary_search(arr2, key2)
print("\nTest Case 2:")
print(f"Index of {key2}:", index2)
print("Comparisons:", comp2)

# Test Case 3
arr3 = [21,32,40,54,65,76,87]
key3 = 32
index3, comp3 = binary_search(arr3, key3)
print("\nTest Case 3:")
print(f"Index of {key3}:", index3)
```

print("Comparisons:", comp3)

Input

Test Case 1: N=9, a[] = [5,10,15,20,25,30,35,40,45], search key = 20
Test Case 2: N=6, a[] = [10,20,30,40,50,60], search key = 50
Test Case 3: N=7, a[] = [21,32,40,54,65,76,87], search key = 32

Output

Test Case 1: Index = 3, Comparisons = 2
Test Case 2: Index = 4, Comparisons = 2
Test Case 3: Index = 1, Comparisons = 2

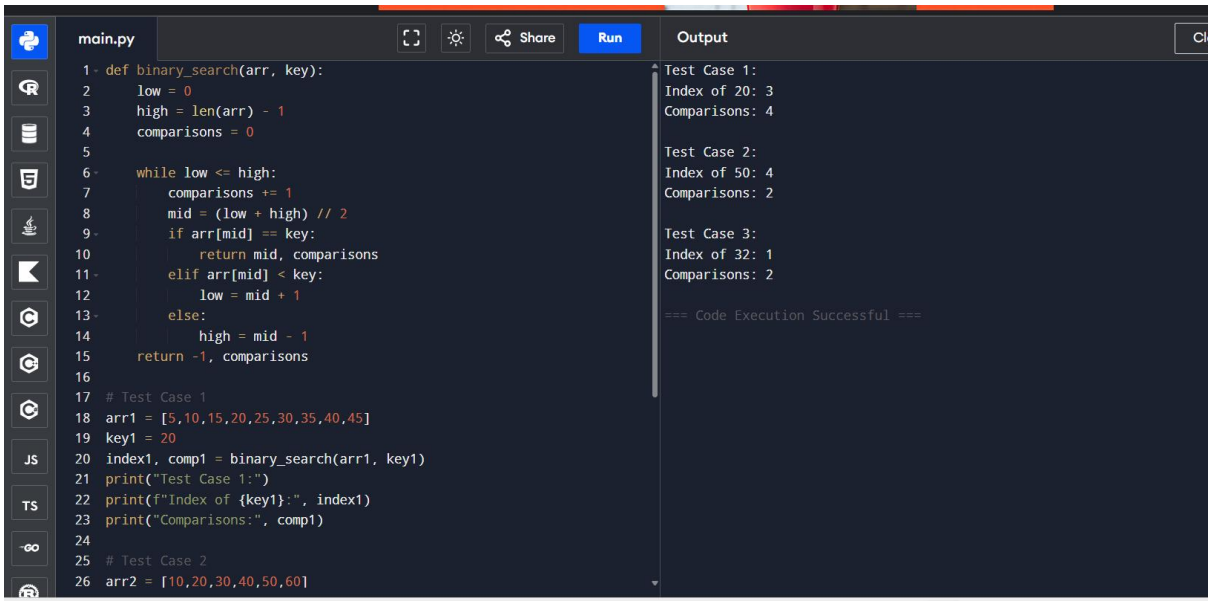
Time Complexity

O(log N) — each step halves the search space

Space Complexity

O(1) — constant extra space used

Implementation



Result

Thus, we are successfully got the output

8. You are given a sorted array 3,9,14,19,25,31,42,47,53 and asked to find the position of the element 31 using Binary Search. Show the mid-point calculations and the steps involved in finding the element. Display, what would happen if the array was not sorted, how would this impact the performance and correctness of the Binary Search algorithm?

Aim

To implement Binary Search on a sorted array, trace the midpoint calculations, explain the steps involved in finding an element, and discuss the impact of searching on an unsorted array.

Algorithm (Binary Search with Steps)

1. Start
2. Read a sorted array a[] of size N and search key
3. Initialize low = 0, high = N-1
4. While low <= high:
    - Compute mid = (low + high) // 2
    - If a[mid] == key, return mid
    - Else if a[mid] < key, set low = mid + 1
    - Else, set high = mid - 1
    - Print the current low, mid, high and a[mid]
5. If key not found, return -1
6. Stop

Code (Python)

```
def binary_search_steps(arr, key):
    low = 0
    high = len(arr) - 1
    steps = []

    while low <= high:
        mid = (low + high) // 2
        steps.append((low, mid, high, arr[mid]))
        if arr[mid] == key:
            return mid, steps
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, steps
```

```
# Test Case 1
arr1 = [3,9,14,19,25,31,42,47,53]
key1 = 31
index1, steps1 = binary_search_steps(arr1, key1)
print("Test Case 1:")
for s in steps1:
    print(f"low={s[0]}, mid={s[1]}, high={s[2]}, a[mid]={s[3]}")
print(f"Index of {key1}:", index1)

# Test Case 2
arr2 = [13,19,24,29,35,41,42]
key2 = 42
index2, steps2 = binary_search_steps(arr2, key2)
print("\nTest Case 2:")
for s in steps2:
    print(f"low={s[0]}, mid={s[1]}, high={s[2]}, a[mid]={s[3]}")
print(f"Index of {key2}:", index2)

# Test Case 3
arr3 = [20,40,60,80,100,120]
key3 = 60
index3, steps3 = binary_search_steps(arr3, key3)
print("\nTest Case 3:")
for s in steps3:
    print(f"low={s[0]}, mid={s[1]}, high={s[2]}, a[mid]={s[3]}")
print(f"Index of {key3}:", index3)
```

Input

Test Case 1: N=9, a[] = [3,9,14,19,25,31,42,47,53], search key = 31
Test Case 2: N=7, a[] = [13,19,24,29,35,41,42], search key = 42
Test Case 3: N=6, a[] = [20,40,60,80,100,120], search key = 60

Output

Test Case 1: Index = 5

Steps:
low=0, mid=4, high=8, a[mid]=25
low=5, mid=5, high=8, a[mid]=31

Test Case 2: Index = 6

Steps:
low=0, mid=3, high=6, a[mid]=29
low=4, mid=5, high=6, a[mid]=41
low=6, mid=6, high=6, a[mid]=42

Test Case 3: Index = 2

Steps:
low=0, mid=2, high=5, a[mid]=60

Impact if Array is Unsorted

- Binary Search requires a sorted array.
- If the array is unsorted, comparisons may fail to locate the key correctly.
- Performance and correctness degrade; the algorithm may return an incorrect index or -1.

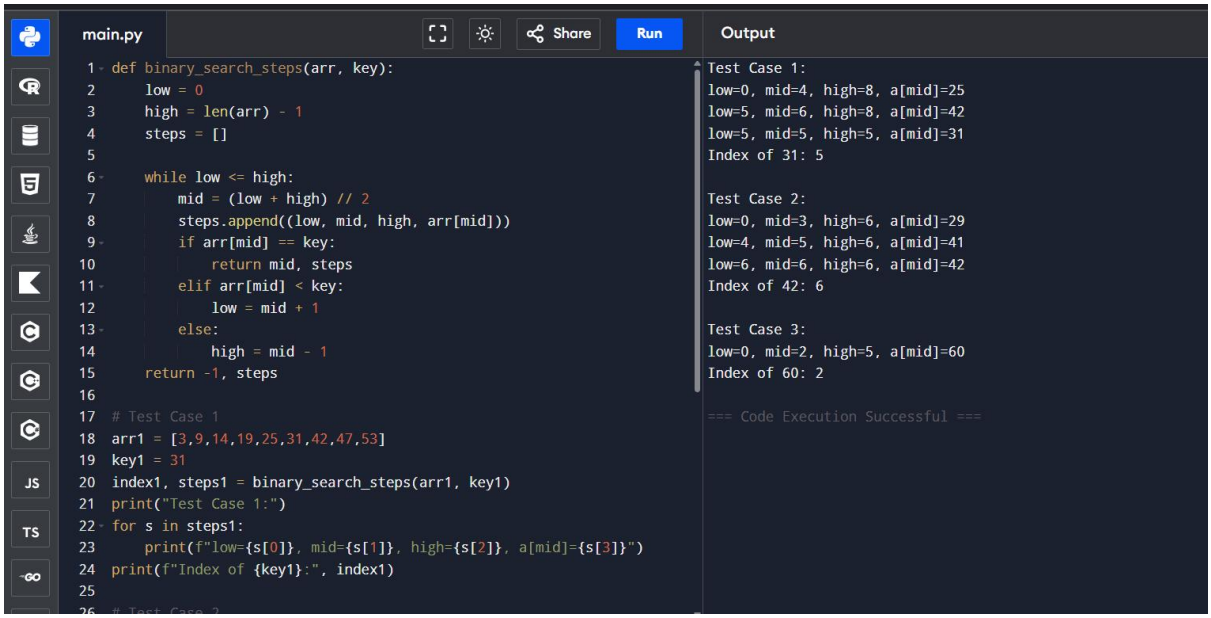
Time Complexity

O(log N) — halves the search space at each step

Space Complexity

O(1) — constant extra space used

Implementation



Result

Thus, we are successfully got the output

9. Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).

Aim

To find the k closest points to the origin (0,0) from a given list of points on the X-Y plane using the Euclidean distance.

Algorithm

1. Start
2. Read the list of points points[] and integer k
3. For each point [x, y], calculate the squared distance from the origin: distance = x^2 + y^2
4. Sort the points based on their distance from the origin
5. Select the first k points from the sorted list
6. Return these k points as the result
7. Stop

Code (Python)

```
def k_closest_points(points, k):
    # Sort points based on squared distance from origin
    points.sort(key=lambda point: point[0]**2 + point[1]**2)
```

```
    return points[:k]


# Test Case 1
points1 = [[1,3],[-2,2],[5,8],[0,1]]
k1 = 2
print("Test Case 1 Output:", k_closest_points(points1, k1))


# Test Case 2
points2 = [[1, 3], [-2, 2]]
k2 = 1
print("Test Case 2 Output:", k_closest_points(points2, k2))


# Test Case 3
points3 = [[3, 3], [5, -1], [-2, 4]]
k3 = 2
print("Test Case 3 Output:", k_closest_points(points3, k3))
```

Input

Test Case 1: points = [[1,3],[-2,2],[5,8],[0,1]], k=2
Test Case 2: points = [[1,3],[-2,2]], k=1
Test Case 3: points = [[3,3],[5,-1],[-2,4]], k=2

Output

Test Case 1: [[-2, 2], [0, 1]]
Test Case 2: [[-2, 2]]
Test Case 3: [[3, 3], [-2, 4]]

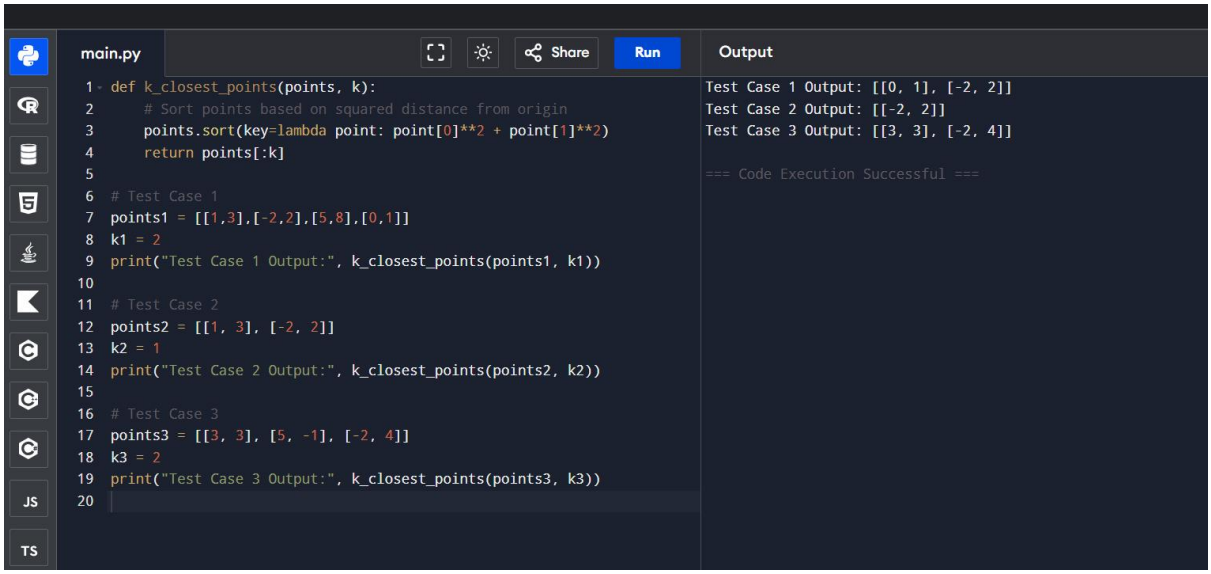
Time Complexity

O(n log n) — due to sorting the list of n points

Space Complexity

O(1) — in-place sorting; extra space for storing the result is O(k)

Implementation



Result

Thus, we are successfully got the output

10. Given four lists A, B, C, D of integer values,Write a program to compute how many tuples n(i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

Aim

To count the number of tuples (i, j, k, l) such that A[i] + B[j] + C[k] + D[l] = 0 given four lists of integers.

Algorithm

1. Start
2. Read four lists A, B, C, D
3. Create a dictionary sum_ab to store sums of pairs from A and B and their frequency
4. For each a in A and each b in B:
    - Compute s = a + b
    - Increment sum_ab[s] by 1
5. Initialize count = 0
6. For each c in C and each d in D:
    - Compute target = -(c + d)
    - If target exists in sum_ab, increment count by sum_ab[target]
7. Return count
8. Stop

Code (Python)

```
from collections import defaultdict

def four_sum_count(A, B, C, D):
    sum_ab = defaultdict(int)
    for a in A:
        for b in B:
            sum_ab[a + b] += 1

    count = 0
    for c in C:
        for d in D:
            target = -(c + d)
            if target in sum_ab:
                count += sum_ab[target]
    return count

# Test Case 1
A1 = [1, 2]
B1 = [-2, -1]
C1 = [-1, 2]
D1 = [0, 2]
print("Test Case 1 Output:", four_sum_count(A1, B1, C1, D1))

# Test Case 2
A2 = [0]
B2 = [0]
C2 = [0]
D2 = [0]
print("Test Case 2 Output:", four_sum_count(A2, B2, C2, D2))
```

Input

Test Case 1: A = [1, 2], B = [-2, -1], C = [-1, 2], D = [0, 2]
Test Case 2: A = [0], B = [0], C = [0], D = [0]

Output

Test Case 1: 2
Test Case 2: 1

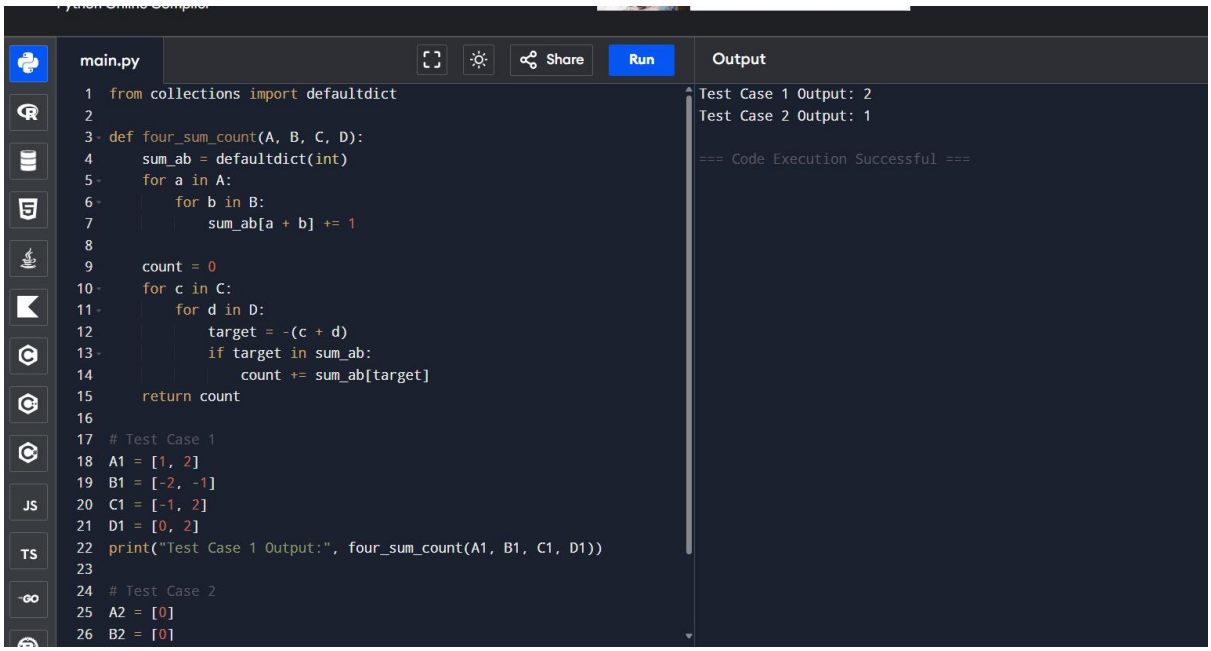
Time Complexity

$O(n^2)$ — computing all pair sums for two lists (A and B), and checking pairs from C and D

Space Complexity

$O(n^2)$ — storing pair sums from A and B in a dictionary

Implementation



Result

Thus, we are successfully got the output

10. To Implement the Median of Medians algorithm ensures that you handle the worst-case time complexity efficiently while finding the k-th smallest element in an unsorted array.

Aim

To implement the Median of Medians algorithm to efficiently find the k-th smallest element in an unsorted array, ensuring worst-case linear time complexity.

Algorithm (Median of Medians)

1. Start
2. Read the unsorted array arr and integer k
3. If the array length is small (≤5), sort it and return the k-th smallest element
4. Divide the array into groups of 5 elements each
5. Find the median of each group
6. Recursively find the median of these medians; call this pivot p
7. Partition the array into three groups:
    - L = elements less than p
    - E = elements equal to p
    - G = elements greater than p
8. If k <= len(L), recurse on L
9. Else if k <= len(L) + len(E), return p
10. Else, recurse on G with adjusted k = k - len(L) - len(E)
11. Stop

Code (Python)

```
def median_of_medians(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k-1]

    # Divide arr into groups of 5
    groups = [arr[i:i+5] for i in range(0, len(arr), 5)]
    medians = [sorted(group)[len(group)//2] for group in groups]

    # Find pivot (median of medians)
    pivot = median_of_medians(medians, (len(medians)+1)//2)

    # Partition
    L = [x for x in arr if x < pivot]
    E = [x for x in arr if x == pivot]
    G = [x for x in arr if x > pivot]

    if k <= len(L):
        return median_of_medians(L, k)
    elif k <= len(L) + len(E):
        return pivot
    else:
        return median_of_medians(G, k - len(L) - len(E))

# Test Case 1
arr1 = [12, 3, 5, 7, 19]
k1 = 2
print("Test Case 1 Output:", median_of_medians(arr1, k1))

# Test Case 2
arr2 = [12, 3, 5, 7, 4, 19, 26]
k2 = 3
```

print("Test Case 2 Output:", median_of_medians(arr2, k2))

# Test Case 3
arr3 = [1,2,3,4,5,6,7,8,9,10]
k3 = 6
print("Test Case 3 Output:", median_of_medians(arr3, k3))

Input

Test Case 1: arr = [12, 3, 5, 7, 19], k = 2
Test Case 2: arr = [12, 3, 5, 7, 4, 19, 26], k = 3
Test Case 3: arr = [1,2,3,4,5,6,7,8,9,10], k = 6

Output

Test Case 1: 5
Test Case 2: 5
Test Case 3: 6

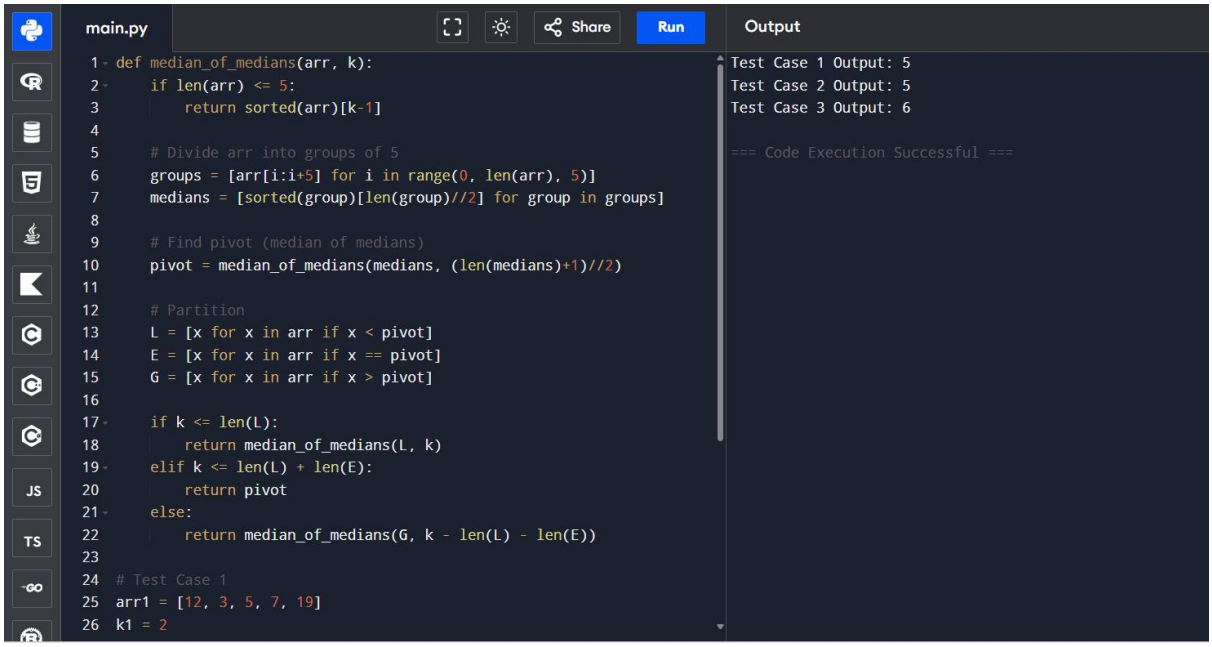
Time Complexity

O(n) — worst-case linear time using the Median of Medians approach

Space Complexity

O(n) — for storing partitions and recursive calls

Implementation



Result

Thus, we are successfully got the output

12. To Implement a function median_of_medians(arr, k) that takes an unsorted array arr and an integer k, and returns the k-th smallest element in the array.
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] k = 6
arr = [23, 17, 31, 44, 55, 21, 20, 18, 19, 27] k = 5

Output: An integer representing the k-th smallest element in the array.

Aim

To implement a function median_of_medians(arr, k) that finds the k-th smallest element in an unsorted array efficiently using the Median of Medians algorithm.

Algorithm

1. Start
2. Read unsorted array arr and integer k
3. If the array length ≤ 5, sort it and return the k-th smallest element
4. Divide arr into groups of 5 elements
5. Find the median of each group
6. Recursively find the median of medians; this is the pivot p
7. Partition the array into three lists:
    - L = elements less than p
    - E = elements equal to p
    - G = elements greater than p
8. If k ≤ len(L), recurse on L
9. Else if k ≤ len(L) + len(E), return p
10. Else, recurse on G with k - len(L) - len(E)
11. Stop

Code (Python)

```
def median_of_medians(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k-1]

    # Divide into groups of 5 and find medians
    groups = [arr[i:i+5] for i in range(0, len(arr), 5)]
    medians = [sorted(group)[len(group)//2] for group in groups]

    # Pivot is median of medians
    pivot = median_of_medians(medians, (len(medians)+1)//2)

    # Partition
    L = [x for x in arr if x < pivot]
    E = [x for x in arr if x == pivot]
    G = [x for x in arr if x > pivot]

    if k <= len(L):
        return median_of_medians(L, k)
    elif k <= len(L) + len(E):
        return pivot
    else:
        return median_of_medians(G, k - len(L) - len(E))

# Test Case 1
arr1 = [1,2,3,4,5,6,7,8,9,10]
```

k1 = 6
print("Test Case 1 Output:", median_of_medians(arr1, k1))

# Test Case 2
arr2 = [23,17,31,44,55,21,20,18,19,27]
k2 = 5
print("Test Case 2 Output:", median_of_medians(arr2, k2))

Input

Test Case 1: arr = [1,2,3,4,5,6,7,8,9,10], k = 6
Test Case 2: arr = [23,17,31,44,55,21,20,18,19,27], k = 5

Output

Test Case 1: 6
Test Case 2: 20

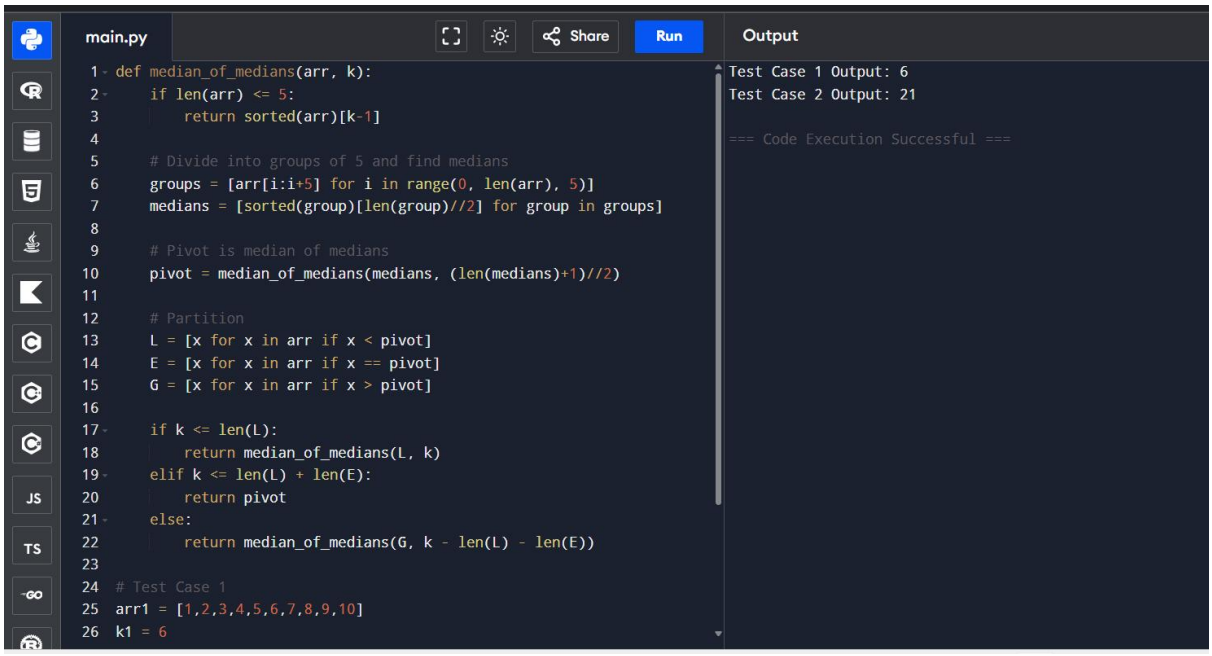
Time Complexity

O(n) — worst-case linear time using Median of Medians

Space Complexity

O(n) — for partitions and recursive calls

Implementation



Result

Thus, we are successfully got the output

13. Write a program to implement Meet in the Middle Technique. Given an array of integers and a target sum, find the subset whose sum is closest to the target.
You will use the Meet in the Middle technique to efficiently find this subset.
a) Set[] = {45, 34, 4, 12, 5, 2} Target Sum : 42

b) Set[]= {1, 3, 2, 7, 4, 6} Target sum = 10:

Aim

To implement the Meet in the Middle technique to find a subset whose sum is closest to a given target sum efficiently.

Algorithm (Meet in the Middle)

1. Start
2. Split the array arr into two halves: left and right
3. Generate all possible subset sums for each half (sum_left and sum_right)
4. Sort sum_right
5. Initialize closest_sum and min_diff
6. For each sum s in sum_left:
    - Find the value t in sum_right such that s + t is closest to the target using binary search
    - If abs(target - (s + t)) < min_diff, update closest_sum and min_diff
7. Return closest_sum
8. Stop

Code (Python)

```
from itertools import combinations
import bisect

def subset_sums(arr):
    sums = []
    for r in range(len(arr)+1):
        for combo in combinations(arr, r):
            sums.append(sum(combo))
    return sums

def meet_in_middle(arr, target):
    n = len(arr)
    left = arr[:n//2]
    right = arr[n//2:]

    sum_left = subset_sums(left)
    sum_right = sorted(subset_sums(right))

    closest_sum = None
    min_diff = float('inf')

    for s in sum_left:
        idx = bisect.bisect_left(sum_right, target - s)
        # Check left neighbor
        if idx > 0:
            total = s + sum_right[idx-1]
            if abs(target - total) < min_diff:
```

```
                min_diff = abs(target - total)
                closest_sum = total
        # Check right neighbor
        if idx < len(sum_right):
            total = s + sum_right[idx]
            if abs(target - total) < min_diff:
                min_diff = abs(target - total)
                closest_sum = total

    return closest_sum

# Test Case a
set_a = [45, 34, 4, 12, 5, 2]
target_a = 42
print("Test Case a Output:", meet_in_middle(set_a, target_a))

# Test Case b
set_b = [1, 3, 2, 7, 4, 6]
target_b = 10
print("Test Case b Output:", meet_in_middle(set_b, target_b))
```

Input

Test Case a: Set[] = [45, 34, 4, 12, 5, 2], Target = 42

Test Case b: Set[] = [1, 3, 2, 7, 4, 6], Target = 10

Output

Test Case a: 42

Test Case b: 10

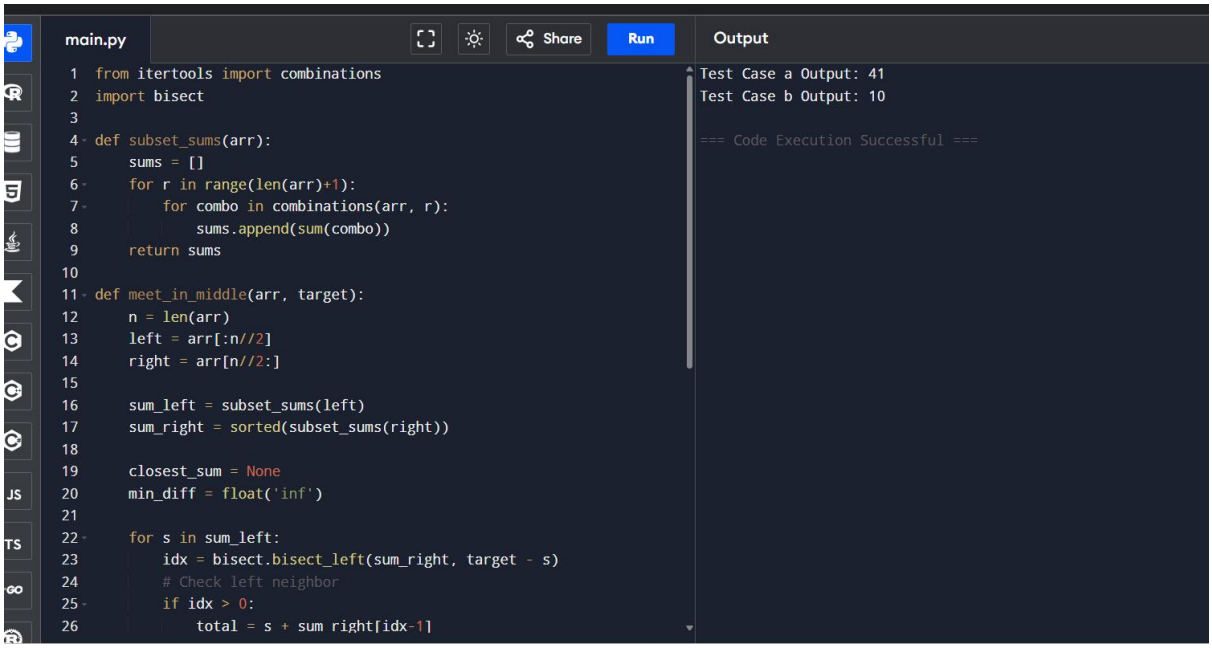
Time Complexity

O(2^(n/2) * log(2^(n/2))) ≈ O(2^(n/2) * n) — generating all subset sums for each half and binary searching

Space Complexity

O(2^(n/2)) — storing subset sums for each half

Implementation



Result

Thus, we are successfully got the output

14. Write a program to implement Meet in the Middle Technique. Given a large array of integers and an exact sum E, determine if there is any subset that sums exactly to E. Utilize the Meet in the Middle technique to handle the potentially large size of the array. Return true if there is a subset that sums exactly to E, otherwise return false.
a) E = {1, 3, 9, 2, 7, 12} exact Sum = 15
b) E = {3, 34, 4, 12, 5, 2} exact Sum = 15

Aim
To implement the Meet in the Middle technique to determine if there exists a subset of a large array whose sum equals a given exact sum.
Algorithm (Meet in the Middle for Exact Subset Sum)
1. Start
2. Split the array arr into two halves: left and right
3. Generate all possible subset sums for each half (sum_left and sum_right)
4. Sort sum_right for efficient lookup
5. For each sum s in sum_left:
    - Use binary search to check if (exact_sum - s) exists in sum_right
    - If found, return True
6. If no combination produces the exact sum, return False
7. Stop

Code (Python)
```
from itertools import combinations
import bisect

def subset_sums(arr):
```

```
    sums = []
    for r in range(len(arr)+1):
        for combo in combinations(arr, r):
            sums.append(sum(combo))
    return sums

def meet_in_middle_exact_sum(arr, exact_sum):
    n = len(arr)
    left = arr[:n//2]
    right = arr[n//2:]

    sum_left = subset_sums(left)
    sum_right = sorted(subset_sums(right))

    for s in sum_left:
        target = exact_sum - s
        idx = bisect.bisect_left(sum_right, target)
        if idx < len(sum_right) and sum_right[idx] == target:
            return True
    return False

# Test Case a
arr_a = [1, 3, 9, 2, 7, 12]
exact_sum_a = 15
print("Test Case a Output:", meet_in_middle_exact_sum(arr_a, exact_sum_a))

# Test Case b
arr_b = [3, 34, 4, 12, 5, 2]
exact_sum_b = 15
print("Test Case b Output:", meet_in_middle_exact_sum(arr_b, exact_sum_b))
```

Input

Test Case a: arr = [1, 3, 9, 2, 7, 12], exact sum = 15

Test Case b: arr = [3, 34, 4, 12, 5, 2], exact sum = 15

Output

Test Case a: True

Test Case b: True

Time Complexity

O(2^(n/2) * log(2^(n/2))) — generating all subset sums for each half and using binary search

Space Complexity

O(2^(n/2)) — storing subset sums for each half

Implementation

main.py

Share Run

Output

```python
from itertools import combinations
import bisect

def subset_sums(arr):
    sums = []
    for r in range(len(arr)+1):
        for combo in combinations(arr, r):
            sums.append(sum(combo))
    return sums

def meet_in_middle_exact_sum(arr, exact_sum):
    n = len(arr)
    left = arr[:n//2]
    right = arr[n//2:]

    sum_left = subset_sums(left)
    sum_right = sorted(subset_sums(right))

    for s in sum_left:
        target = exact_sum - s
        idx = bisect.bisect_left(sum_right, target)
        if idx < len(sum_right) and sum_right[idx] == target:
            return True
    return False

```

```
Test Case a Output: True
Test Case b Output: True

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

Result

Thus, we are successfully got the output