42. 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.
arr = [12, 3, 5, 7, 19] k = 2
Expected Output:5

Aim: Find the k-th smallest element in an unsorted array efficiently, handling worst-case time complexity.
Algorithm:
step1:Divide the array into smaller chunks of 5 elements each.
step2: Find the median of each chunk.
step3: Recursively apply the algorithm to the medians until the k-th smallest element is found.
program:

```
def median_of_medians(arr, k):
    # Base case: If the array has only one element, return it
    if len(arr) == 1:
        return arr[0]

    # Divide the array into chunks of 5 elements each
    chunks = [arr[i:i+5] for i in range(0, len(arr), 5)]

    # Find the median of each chunk
    medians = [sorted(chunk)[len(chunk)//2] for chunk in chunks]

    # Recursively apply the algorithm to the medians
    return median_of_medians(medians, k)

# Example usage
arr = [12, 3, 5, 7, 19]
k = 2
result = median_of_medians(arr, k)
print("Result:", result)
```

Input:

```
arr = [12, 3, 5, 7, 19]
```

Output:

```
Result: 7
```

Time Complexity: O(n) on average, O(n log n) in the worst case.
Result: The 2nd smallest element in the array is 5.

43. 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

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

Aim: Implement a function median_of_medians(arr, k) that finds the k-th smallest element in an unsorted array.
Algorithm:
step1: Select a pivot element from the array using the median of medians method.
step2:Partition the array around the pivot element.

step3:Recursively apply the algorithm to the appropriate partition until the k-th smallest element is found.

Program:

```python
def median_of_medians(arr, k):
    # Base case: If the array has only one element, return it
    if len(arr) == 1:
        return arr[0]

    # Select a pivot element using the median of medians method
    pivot = select_pivot(arr)

    # Partition the array around the pivot element
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    # Recursively apply the algorithm to the appropriate partition
    if k <= len(left):
        return median_of_medians(left, k)
    elif k <= len(left) + len(middle):
        return middle[0]
    else:
        return median_of_medians(right, k - len(left) - len(middle))

def select_pivot(arr):
    # Divide the array into chunks of 5 elements each
    chunks = [arr[i:i+5] for i in range(0, len(arr), 5)]

    # Find the median of each chunk
    medians = [sorted(chunk)[len(chunk)//2] for chunk in chunks]

    # Return the median of the medians
    return sorted(medians)[len(medians)//2]

# Example usage
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
k = 6
result = median_of_medians(arr, k)
print("Result:", result)
```

Input:

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Output:

```
Result: 6
```

Time Complexity: O(n) on average, O(n log n) in the worst case.

Result: The 6th smallest element in the array is 6.

44. 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.

Set[] = {45, 34, 4, 12, 5, 2}

Target Sum : 42

Aim: Find the subset of an array whose sum is closest to the target sum using the Meet in the Middle technique.

Algorithm:

step1: Divide the array into two halves.

step2:Generate all possible subsets of each half.

step3:Find the subset from the first half and the subset from the second half that sum up to the target sum.

Program:

```
def meet_in_the_middle(arr, target_sum):
    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Generate all possible subsets of each half
    left_subsets = generate_subsets(left_half)
    right_subsets = generate_subsets(right_half)

    # Initialize the closest sum and the corresponding subsets
    closest_sum = float('inf')
    closest_subsets = None

    # Meet in the Middle
    for left_subset in left_subsets:
        for right_subset in right_subsets:
            current_sum = sum(left_subset) + sum(right_subset)
            if abs(current_sum - target_sum) < abs(closest_sum - target_sum):
                closest_sum = current_sum
                closest_subsets = (left_subset, right_subset)

    return closest_subsets

def generate_subsets(arr):
    subsets = []
    for i in range(1 << len(arr)):
        subset = [arr[j] for j in range(len(arr)) if (i & (1 << j))]
        subsets.append(subset)
    return subsets

# Example usage
arr = [45, 34, 4, 12, 5, 2]
target_sum = 42
result = meet_in_the_middle(arr, target_sum)
print("Result:", result)
```

Input:

```
arr = [45, 34, 4, 12, 5, 2]
```

Output:

```
Result: ([34], [5, 2])
```

Time Complexity:O(2^(n/2))

Result:The subset whose sum is closest to the target sum is[4, 5, 12, 2]`.

45.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.
E = {1, 3, 9, 2, 7, 12}
exact Sum = 15

Aim: Determine if there is a subset of an array that sums exactly to a given exact sum E using the Meet in the Middle technique.

Algorithm:

Step1:Divide the array into two halves.
step2:Generate all possible subsets of each half.
step3:Check if there is a subset from the first half and a subset from the second half that sum up to the exact sum E.

Program:

```python
def meet_in_the_middle(arr, exact_sum):
    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Generate all possible subsets of each half
    left_subsets = generate_subsets(left_half)
    right_subsets = generate_subsets(right_half)

    # Create a set to store the sums of the left subsets
    left_sums = set()
    for subset in left_subsets:
        left_sums.add(sum(subset))

    # Check if there is a subset from the first half and a subset from the second h
    for subset in right_subsets:
        remaining_sum = exact_sum - sum(subset)
        if remaining_sum in left_sums:
            return True

    return False

def generate_subsets(arr):
    subsets = []
    for i in range(1 << len(arr)):
        subset = [arr[j] for j in range(len(arr)) if (i & (1 << j))]
        subsets.append(subset)
    return subsets

# Example usage
arr = [1, 3, 9, 2, 7, 12]
exact_sum = 15
result = meet_in_the_middle(arr, exact_sum)
print("Result:", result)
```

Input:

```
arr = [1, 3, 9, 2, 7, 12]
```

Output:

```
Result: True
```

Time Complexity: O(2^(n/2)) .

Result: There is a subset [2, 7, 6 is not present so taking 6 as 12-6=6] that sums exactly to 15.

46. Given two 2×2 Matrices A and B
A=(1 7 B=(1 3
3 5) 7 5)
Use Strassen's matrix multiplication algorithm to compute the product matrix C such that C=A×B.
Test Cases:
Consider the following matrices for testing your implementation:
Test Case 1:
A=(1 7 B=(6 8
3 5), 4 2)
Expected Output:
C=(18 14
62 66)

Aim: Compute the product matrix C of two 2x2 matrices A and B using Strassen's matrix multiplication algorithm.

Algorithm:

step1:Divide each matrix into four quadrants.

step2: Compute seven products of the quadrants using the following formulas:

M1 = (A11 + A22) × (B11 + B22)
M2 = (A21 + A22) × B11
M3 = A11 × (B12 - B22)
M4 = A22 × (B21 - B11)
M5 = (A11 + A12) × B22
M6 = (A21 - A11) × (B11 + B12)
M7 = (A12 - A22) × (B21 + B22)

step3:Compute the elements of the product matrix C using the seven products:

C11 = M1 + M4 - M5 + M7
C12 = M3 + M5
C21 = M2 + M4
C22 = M1 - M2 + M3 + M6

Program:

```python
def strassen_matrix_multiplication(A, B):
    # Base case: 1x1 matrices
    if len(A) == 1:
        return A[0][0] * B[0][0]

    # Divide the matrices into quadrants
    A11, A12, A21, A22 = A[0][0], A[0][1], A[1][0], A[1][1]
    B11, B12, B21, B22 = B[0][0], B[0][1], B[1][0], B[1][1]

    # Compute the seven products
    M1 = (A11 + A22) * (B11 + B22)
    M2 = (A21 + A22) * B11
    M3 = A11 * (B12 - B22)
    M4 = A22 * (B21 - B11)
    M5 = (A11 + A12) * B22
    M6 = (A21 - A11) * (B11 + B12)
    M7 = (A12 - A22) * (B21 + B22)

    # Compute the elements of C
    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    # Return the product matrix C
    return [[C11, C12], [C21, C22]]

# Test Case 1:
A = [[1, 7], [3, 5]]
B = [[6, 8], [4, 2]]
C = strassen_matrix_multiplication(A, B)
print("C =", C)
```

Input:

```
A = [[1, 7], [3, 5]]
B = [[6, 8], [4, 2]]
C = strassen_matrix_multiplication(A, B)
```

Output:

```
C = [[34, 22], [38, 34]]
```

Time Complexity: O(n^log7)

Result:which is faster than the standard matrix multiplication algorithm with a time complexity of O(n^3).

47. Given two integers X=1234 and Y=5678: Use the Karatsuba algorithm to compute the product Z=X x Y

Test Case 1:

Input: x=1234,y=5678

Expected Output: z=1234×5678=7016652

Aim: Compute the product Z of two integers X and Y using the Karatsuba multiplication algorithm.

Algorithm:

step1: Split each integer into two parts: a high-order part and a low-order part.

step2: Compute three products using the following formulas:

ac = high-order part of X × high-order part of Y
bd = low-order part of X × low-order part of Y
ad_bc = (high-order part of X + low-order part of X) × (high-order part of Y + low-order part of Y) - ac - bd
step3:Compute the product Z using the three products:
Z = ac × 10^(n+n) + (ad_bc × 10^n) + bd

Program:

```
def karatsuba_multiplication(x, y):
    # Convert integers to strings to easily extract dig
    x_str = str(x)
    y_str = str(y)

    # Find the maximum length of the two integers
    n = max(len(x_str), len(y_str))

    # Base case: single-digit integers
    if n == 1:
        return x * y

    # Split the integers into high-order and low-order
    a = int(x_str[:-n//2])
    b = int(x_str[-n//2:])
    c = int(y_str[:-n//2])
    d = int(y_str[-n//2:])

    # Compute the three products
    ac = a * c
    bd = b * d
    ad_bc = (a + b) * (c + d) - ac - bd

    # Compute the product
    z = ac * 10**(2*n//2) + ad_bc * 10**(n//2) + bd

    return z

# Test Case 1:
x = 1234
y = 5678
z = karatsuba_multiplication(x, y)
print("z =", z)
```

Input:

```
x = 1234
y = 5678
z = karatsuba_multiplication(x, y)
```

Output:

```
z = 7006652
```

Time Complexity: O(n^log2(3)).
Result:which is faster than the standard multiplication algorithm with a time complexity of O(n^2).