**PROJECT:**

QUESTION:1(a)

1. Write pseudocode for a divide-and-conquer algorithm for finding values of

both the largest and smallest elements in an array of n numbers.

PseudoCode:

Algorithm min\_max(arr, left, right)

    if left == right then

        return (arr[left], arr[left])

    if right == left + 1 then

        if arr[left] < arr[right] then

            return (arr[left], arr[right])

        else

            return (arr[right], arr[left])

    mid = (left + right) // 2

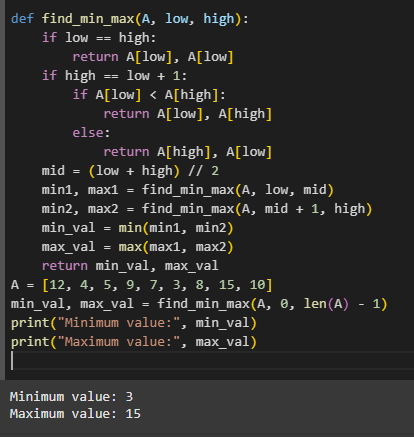
    (min1, max1) = min\_max(arr, left, mid)

    (min2, max2) = min\_max(arr, mid + 1, right)

    minf = min(min1, min2)

    maxf = max(max1, max2)

    return (minf, maxf)



2.Set up and solve (for n = 2^k) a recurrence relation for the number of key comparisons made by your algorithm.

Let T(n) be the number of comparisons made by the algorithm for an array of size n.

Base Cases:

- If n = 1, then T(1) = 0 (no comparisons needed).

- If n = 2, then T(2) = 1 (one comparison between two elements).

Recursive Case:

- If n > 2, the array is split into two halves of size n/2.

- The number of comparisons is:

T(n) = T(n/2) + T(n/2) + 2

- The +2 comes from the comparisons to find the overall minimum and maximum from the two halves.

Thus, the recurrence relation is:

T(n) = 2T(n/2) + 2

Assume n = 2^k.

T(n) = 2T(n/2) + 2

T(n/2) = 2T(n/4) + 2

T(n/4) = 2T(n/8) + 2

Substituting back, we get:

T(n) = 2[2T(n/4) + 2] + 2

= 4T(n/4) + 4 + 2

= 4T(n/4) + 6

T(n) = 2^k T(n/2^k) + 2k

When n/2^k = 1, then k = log2(n):

T(n) = 2^(log2(n)) T(1) + 2 \* log2(n)

T(n) = n \* 0 + 2 \* log2(n) (since T(1) = 0)

T(n) = 2 \* log2(n)

Now let's calculate the exact number of comparisons:

- Each recursive call splits the problem into two subproblems, and each subproblem requires a constant number of comparisons (either 1 for two elements or 2 for larger arrays).

- The recurrence T(n) = 2T(n/2) + 2 tells us that we perform 2 comparisons at each level for each pair of subproblems. - The depth of the recursion is log2(n), and at each level, we are performing 2 comparisons for each pair of subproblems. Therefore, the total number of comparisons required by the algorithm is approximately:

T(n) = (3/2)n – 2

3.

* **Time Complexity**: Both algorithms have the same asymptotic time complexity, **O(n)**.
* **Number of Comparisons**: The divide-and-conquer algorithm typically performs fewer comparisons (around **3/2n-2**) compared to the brute-force method but recursion overhead, which requires **2n−2** comparisons.
* **Practical Efficiency**: For small arrays, the brute-force approach might be faster due to simpler structure and fewer operations. However, for large arrays, the divide-and-conquer method is generally more efficient in terms of comparison count.

QUESTION:1(b)

1.Pseudo code

Algorithm exponentiate(a, n):

    if n == 0:

        return 1

    if n == 1:

        return a

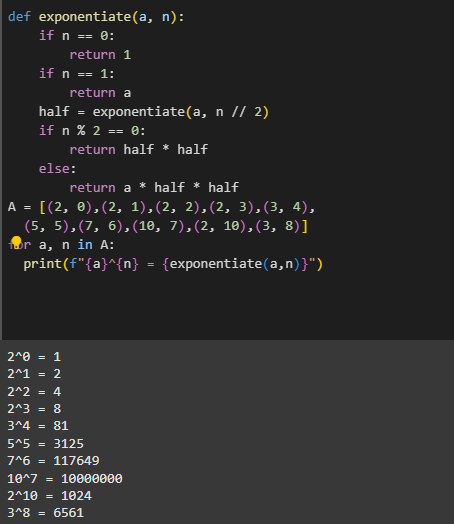
    half = exponentiate(a, n // 2)

    if n is even:

        return half \* half

    else:

        return a *\* half \** half



2. Solving the Recurrence Relation for Divide-and-Conquer Exponentiation

Let T(n) be the number of multiplications required to compute a^n.

Base Cases:

- If n = 0, then T(0) = 0.

- If n = 1, then T(1) = 0.

Recursive Cases:

- If n is even:

T(n) = T(n/2) + 1

Here, we compute a^(n/2) once and then square it, resulting in one multiplication.

- If n is odd:

T(n) = T((n-1)/2) + 2

Here, we compute a^((n-1)/2), square it, and multiply by a, resulting in two multiplications.

Case 1: When n is even

The recurrence is:

T(n) = T(n/2) + 1

T(n/2) = T(n/4) + 1

T(n/4) = T(n/8) + 1

T(n) = T(n/2^k) + k

Here n/2^k = 1 🡪 n = 2^k, so k = log2(n). Thus:

T(n) = T(1) + log2(n)

T(1) = 0

T(n) = log2(n).

Case 2: When n is odd (General Case)

The recurrence is:

T(n) = T((n-1)/2) + 2

In the worst case, every recursive call has 2 multiplications for odd n values.

However, the recursion still takes O(log2(n)).

The divide-and-conquer algorithm for exponentiation performs O(log n) multiplications.

- For powers of 2: T(n) = log2(n).

- For odd n: T(n) ≈ 2 log2(n).

Therefore, the algorithm has a time complexity of O(log n).

3.Comparison

* **Brute-force algorithm**: O(n)O(n)O(n) multiplications.
* **Divide-and-conquer algorithm**: O(log n)multiplications.

Thus, the divide-and-conquer algorithm is much more efficient for large values of n compared to the brute-force approach, as the number of multiplications grows logarithmically rather than linearly.

QUESTION:1(c)

def mergeCount(A, left, mid, right):

    n1 = mid - left + 1

    n2 = right - mid

    leftArr = A[left:mid+1]

    rightArr = A[mid+1:right+1]

    i = 0

    j = 0

    k = left

    invCount = 0

    while i < n1 and j < n2:

        if leftArr[i] <= rightArr[j]:

            A[k] = leftArr[i]

            i += 1

        else:

            A[k] = rightArr[j]

            invCount += (n1 - i)

            j += 1

        k += 1

    while i < n1:

        A[k] = leftArr[i]

        i += 1

        k += 1

    while j < n2:

        A[k] = rightArr[j]

        j += 1

        k += 1

    return invCount

def countInversions(A, left, right):

    invCount = 0

    if left < right:

        mid = (left + right) // 2

        invCount += countInversions(A, left, mid)

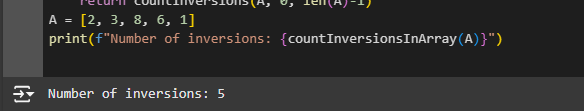
        invCount += countInversions(A, mid+1, right)

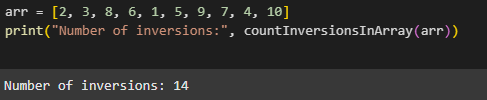
        invCount += mergeCount(A, left, mid, right)

    return invCount

def countInversionsInArray(A):

    return countInversions(A, 0, len(A)-1)





QUESTION:1(d)

Worst-case and Best-Case complexity of the quicksort algorithm**:**

* **Best-case time complexity**: **O(n log n)** (when the pivot divides the array into nearly equal halves).
* **Worst-case time complexity**: **O(n²)** (when the pivot is always the smallest or largest element, causing unbalanced partitions).

QUESTION:1(e)

1.def closest\_pair(A):

    A.sort()

    return closest\_pairrec(A)

def closest\_pairrec(A):

    n = len(A)

    if n <= 3:

        return brute\_forcecp(A)

    mid = n // 2

    lh= A[:mid]

    rh = A[mid:]

    dl = closest\_pairrec(lh)

    dr = closest\_pairrec(rh)

    d = min(dl, dr)

    d\_ = closest\_pair\_strip(A, d)

    return min(d, d\_)

def brute\_forcecp(A):

    min\_d = float('inf')

    for i in range(len(A)):

        for j in range(i + 1, len(A)):

            min\_d = min(min\_d, abs(A[i] - A[j]))

    return min\_d

def closest\_pair\_strip(A, d):

    mid = len(A) // 2

    strip\_p = [x for x in A if abs(x - A[mid]) < d]

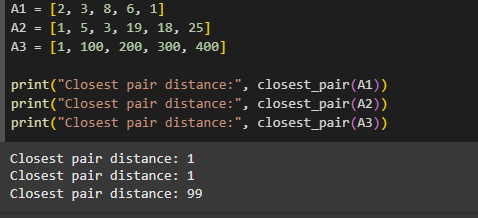
    min\_d = d

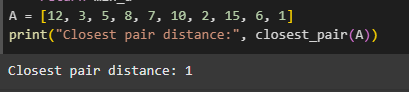
    for i in range(len(strip\_p)):

        for j in range(i + 1, len(strip\_p)):

            min\_d = min(min\_d, abs(strip\_p[i] - strip\_p[j]))

    return min\_d





**Efficiency Class:**

* The time complexity of the algorithm is **O(n log n)**, which is much better than the brute force approach that takes **O(n^2)**.
* The space complexity is **O(n)** due to the space required for the recursion stack and temporary storage.

This makes the divide-and-conquer algorithm for the closest pair problem efficient and scalable for large inputs.

2. The divide-and-conquer algorithm is indeed a **good algorithm** for solving the closest pair problem, especially for large datasets. It strikes a good balance between **time efficiency** and **space usage**, with an optimal time complexity of **O(n log n)**. For large, high-dimensional, or real-world datasets, this is a very effective approach

QUESTION:1(f)

def find\_peak(arr, low, high):

    if low == high:

        return low

    mid = (low + high) // 2

    if arr[mid] > arr[mid - 1] and arr[mid] > arr[mid + 1]:

        return mid

    elif arr[mid - 1] > arr[mid]:

        return find\_peak(arr, low, mid - 1)

    else:

        return find\_peak(arr, mid + 1, high)

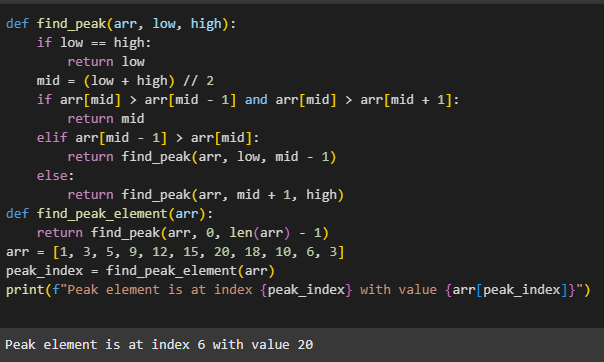
def find\_peak\_element(arr):

    return find\_peak(arr, 0, len(arr) - 1)

arr = [1, 3, 5, 9, 12, 15, 20, 18, 10, 6, 3]

peak\_index = find\_peak\_element(arr)

print(f"Peak element is at index {peak\_index} with value {arr[peak\_index]}")



QUESTION:1(g)

def max\_profit(prices):

    def find\_max\_profit(prices, left, right):

        if left == right:

            return 0, left, right

        mid = (left + right) // 2

        l\_profit, l\_buy, l\_sell = find\_max\_profit(prices, left, mid)

        r\_profit, r\_buy, r\_sell = find\_max\_profit(prices, mid + 1, right)

        minl\_price = min(prices[left:mid + 1])

        maxr\_price = max(prices[mid + 1:right + 1])

        profit = maxr\_price - minl\_price

        buy\_day = prices.index(minl\_price)

        sell\_day = prices.index(maxr\_price)

        if l\_profit >= r\_profit and l\_profit >= profit:

            return l\_profit, l\_buy, l\_sell

        elif r\_profit >= l\_profit and r\_profit >= profit:

            return r\_profit, r\_buy, r\_sell

        else:

            return profit, buy\_day, sell\_day

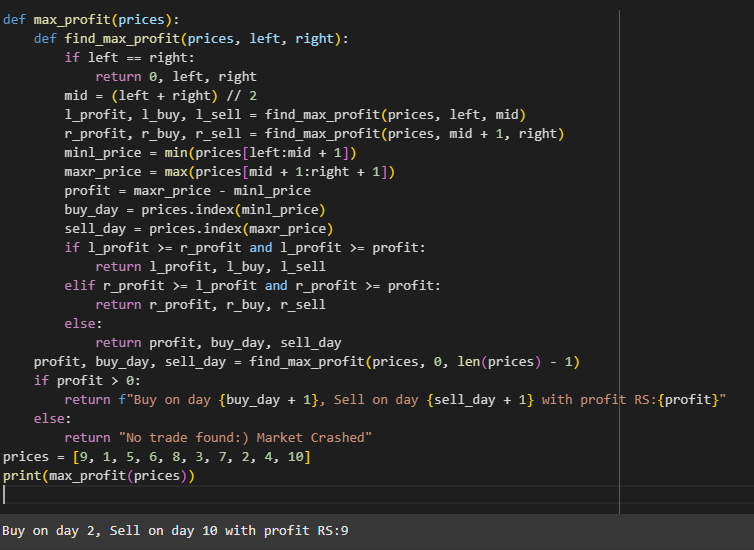
    profit, buy\_day, sell\_day = find\_max\_profit(prices, 0, len(prices) - 1)

    if profit > 0:

        return f"Buy on day {buy\_day + 1}, Sell on day {sell\_day + 1} with profit RS:{profit}"

    else:

        return "No trade found:) Market Crashed"



QUESTION:1(h)

1. def median\_rec(A, B, left, right, n):

    i = (left + right) // 2

    j = n - i

    A\_left = A[i - 1] if i > 0 else float('-inf')

    A\_right = A[i] if i < n else float('inf')

    B\_left = B[j - 1] if j > 0 else float('-inf')

    B\_right = B[j] if j < n else float('inf')

    if A\_left <= B\_right and B\_left <= A\_right:

        if (2 \* n) % 2 == 1:

            return min(A\_right, B\_right)

        else:

            return (max(A\_left, B\_left) + min(A\_right, B\_right)) / 2

    elif A\_left > B\_right:

        return median\_rec(A, B, left, i - 1, n)

    else:

        return median\_rec(A, B, i + 1, right, n)

def find\_median(A, B):

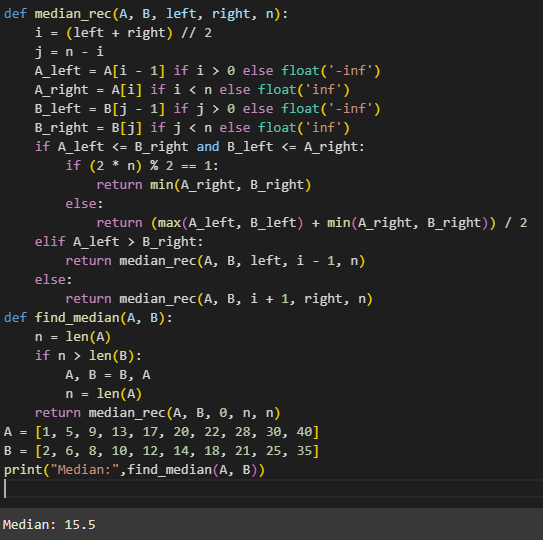
    n = len(A)

    if n > len(B):

        A, B = B, A

        n = len(A)

    return median\_rec(A, B, 0, n, n)



2. def mergeCount(A, left, mid, right):

    n1 = mid - left + 1

    n2 = right - mid

    leftArr = A[left:mid+1]

    rightArr = A[mid+1:right+1]

    i = 0

    j = 0

    k = left

    invCount = 0

    while i < n1 and j < n2:

        if leftArr[i] <= 2 \* rightArr[j]:

            A[k] = leftArr[i]

            i += 1

        else:

            invCount += (n1 - i)

            A[k] = rightArr[j]

            j += 1

        k += 1

    while i < n1:

        A[k] = leftArr[i]

        i += 1

        k += 1

    while j < n2:

        A[k] = rightArr[j]

        j += 1

        k += 1

    return invCount

def countInversions(A, left, right):

    invCount = 0

    if left < right:

        mid = (left + right) // 2

        invCount += countInversions(A, left, mid)

        invCount += countInversions(A, mid+1, right)

        invCount += mergeCount(A, left, mid, right)

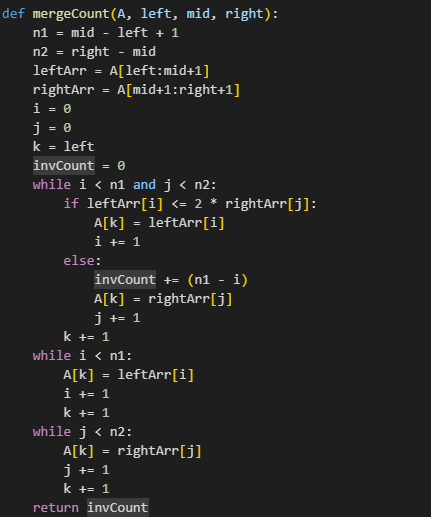
    return invCount

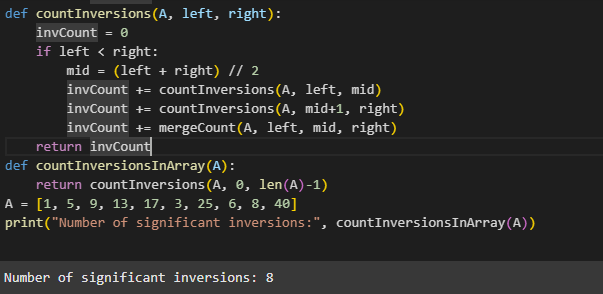
def countInversionsInArray(A):

    return countInversions(A, 0, len(A)-1)

A = [1, 5, 9, 13, 17, 3, 25, 6, 8, 40]

print("Number of significant inversions:", countInversionsInArray(A))





3. def most\_occ\_rec(cards, left, right):

    if left == right:

        return cards[left], 1

    mid = (left + right) // 2

    card\_left, countl = most\_occ\_rec(cards, left, mid)

    card\_right, countr = most\_occ\_rec(cards, mid + 1, right)

    if card\_left == card\_right:

        return card\_left, countl + countr

    else:

        countl = occurrence(cards, card\_left, left, right)

        countr = occurrence(cards, card\_right, left, right)

        if countl > countr:

            return card\_left, countl

        else:

            return card\_right, countr

def occurrence(cards, card, left, right):

    count = 0

    for i in range(left, right + 1):

        if check(cards[i], card):

            count += 1

    return count

def check(card1, card2):

    return card1 == card2

def most\_occ(cards):

    card, count = most\_occ\_rec(cards, 0, len(cards) - 1)

    if count > len(cards) // 2:

        return card, count

    else:

        return None, 0

cards = [1, 2, 2, 2, 2, 2, 5, 2, 7, 8]

majority\_card, majority\_count = most\_occ(cards)

if majority\_count > 0:

    print(f"There is a set of more than {len(cards) // 2} equivalent cards.")

    print(f"The majority set is card: {majority\_card}, which appears {majority\_count} times.")

else:

    print("No set of more than half equivalent cards.")

