DAA PROJECT DELIVERABLE 1

Group Members:

ASHAR ZAMIR 22K-4241

BILAL AHMED 22K-4525

ANAS SALEEM 22K-0500

Q1)

PART A:

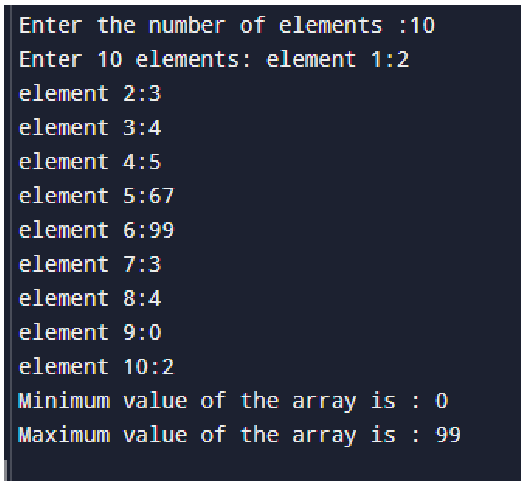
Pseudo-code:

- We will start with the array `Array[L, R]` and variables `min_Value` and `max_Value`.
- First, check if the array has only one element:
- If there is only one element (when `L` is equal to `R`), then that element is both the minimum and maximum. Set `min_Value` and `max_Value` to this element.
- Second, check if the array has two elements:
- If the array has exactly two elements (when `R` is equal to `L + 1`), compare the two elements.
- If the first element is smaller than or equal to the second element:
  - Set `min_Value` to the first element.
  - Set `max_Value` to the second element.
- else:
  - Set `min_Value` to the second element.
  - Set `max_Value` to the first element.
- For arrays (more than two elements):
- Divide the array into two halves:
  - Find the midpoint of the array: `mid = floor((L + R) / 2)`.
- Recursively find the minimum and maximum in the left half (`Array[L, mid]`).
- Recursively find the minimum and maximum in the right half (`Array[mid + 1, R]`).
- After both recursive calls:
- Compare the minimum values from the two halves:
  - If the minimum from the second half is smaller, update `min_Value` to that value.
- Compare the maximum values from the two halves:
  - If the maximum from the second half is larger, update `max_Value` to that value.
- Return the final `min_Value` and `max_Value`.

CODE:

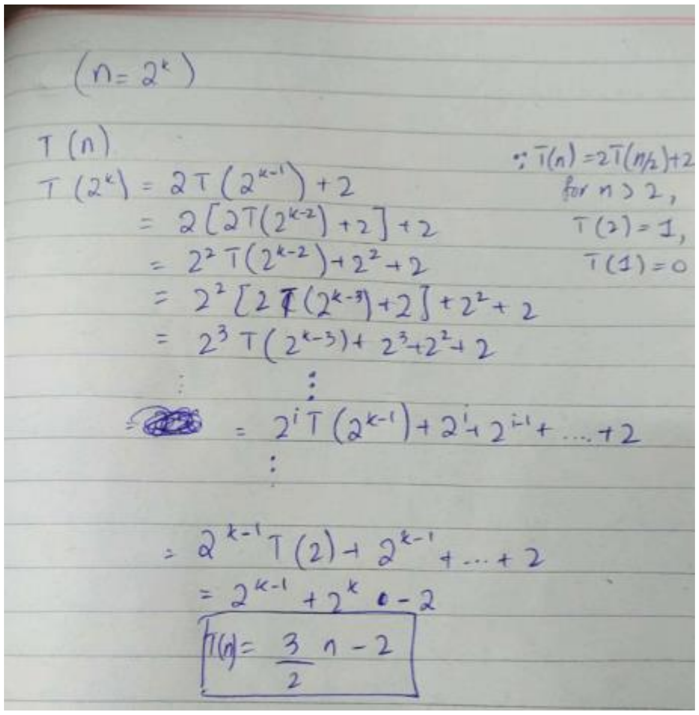
```
#include <iostream>
using namespace std;
void findMinMax(int arr[], int low, int high, int
&minValue, int &maxValue) {
    if (low == high) {
        if (arr[low] < minValue) {
            minValue = arr[low];
        }
        if (arr[low] > maxValue) {
            maxValue = arr[low];
        }
        return;
    }

    if (high == low + 1) {
        if (arr[low] < arr[high]) {
            if (arr[low] < minValue) {
                minValue = arr[low];
            }
            if (arr[high] > maxValue) {
                maxValue = arr[high];
            }
        } else {
            if (arr[high] < minValue) {
                minValue = arr[high];
            }
            if (arr[low] > maxValue) {
                maxValue = arr[low];
            }
        }
        return;
    }

    int mid = (low + high) / 2;

   findMinMax(arr, low, mid, minValue, maxValue);

   findMinMax(arr, mid + 1, high, minValue, maxValue);
}

int main() {
    int n;
    cout << "Enter the number of elements :";
    cin >> n;
    if (n < 10) {
        cout << "Question requirement is atleast 10
elements."<<endl;
    }

    int arr[n];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cout << "element " << i+1 <<":";
        cin >> arr[i];
    }

    int minValue = arr[0];
    int maxValue = arr[0];
    findMinMax(arr, 0, n - 1, minValue, maxValue);
    cout << "Minimum value of the array is : " << minValue
<< endl;
    cout << "Maximum value of the array is : " << maxValue
<< endl;

    return 0;
}
```

OUTPUT:



```
Enter the number of elements :10
Enter 10 elements: element 1:2
element 2:3
element 3:4
element 4:5
element 5:67
element 6:99
element 7:3
element 8:4
element 9:0
element 10:2
Minimum value of the array is : 0
Maximum value of the array is : 99
```

2: for(n=2^k)



$(n = 2^k)$

$\because T(n) = 2T(n/2) + 2$ for $n > 2$, $T(2) = 1$, $T(1) = 0$

$T(n)$

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 2 \\
&= 2[2T(2^{k-2}) + 2] + 2 \\
&= 2^2 T(2^{k-2}) + 2^2 + 2 \\
&= 2^2[2T(2^{k-3}) + 2] + 2^2 + 2 \\
&= 2^3 T(2^{k-3}) + 2^3 + 2^2 + 2 \\
&\vdots \\
&= 2^i T(2^{k-i}) + 2^i + 2^{i-1} + \dots + 2 \\
&\vdots \\
&= 2^{k-1} T(2) + 2^{k-1} + \dots + 2 \\
&= 2^{k-1} + 2^k - 2
\end{aligned}
$$

$$\boxed{T(n) = \frac{3}{2} n - 2}$$

3: Brute Force linear scans the array for comparison of each element whereas this algorithm will take fewer comparisons since we recursively divide the comparison into left and right sub arrays.

PART B:

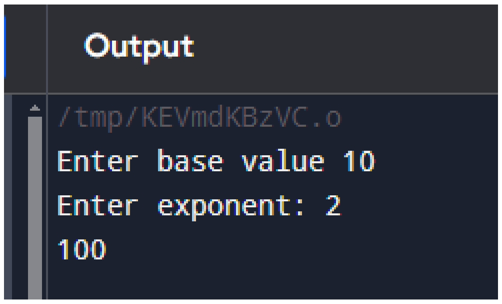
1. Pseudocode:

- Step 1: Base case (when n = 0)
    a. If the exponent n is 0, return 1. This is because any number raised to the power of 0 is always 1.
- Step 2: Check if n is even (n mod 2 = 0):
    a. If n is an even number, calculate the result by squaring the base a (compute a * a) and reduce the exponent by half (use n / 2 as the new exponent).
    b. Recursively call the function with the new base (a * a) and the reduced exponent (n / 2).
- Step 3: Check if n is odd (else case):
    a. If n is an odd number, first multiply the base a by the result of recursively calculating the exponent for a * a with half the value of n (use n / 2 as the new exponent).
    b. This handles the extra factor of a in cases where the exponent is odd.
- Step 4: Return result

CODE:

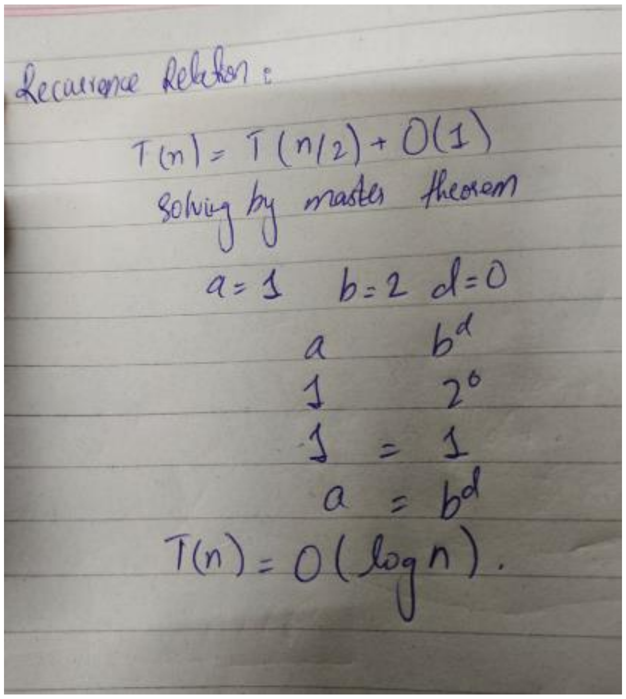
```
#include <iostream>

using namespace std;

int exp_calculation(int a, int n) {

    //checking base casw a^0=1

    if (n == 0)

        return 1;


    // checking even

    if (n % 2 == 0) {

        int half_exp = exp_calculation(a, n / 2);

        return half_exp * half_exp;

    }

    //n is odd

    else {
        int half_exp = exp_calculation(a, (n - 1) / 2);
        return a * half_exp * half_exp;
    }
}

int main() {
    int a;
    int n;
    cout << "Enter base value ";
    cin >> a;
    cout << "Enter exponent: ";
    cin >> n;
    int result = exp_calculation(a, n);
    cout << result << endl;
}
```

OUTPUT:



```
/tmp/KEVmdKBzVC.o
Enter base value 10
Enter exponent: 2
100
```

2:


Recurrence Relation :

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

solving by master theorem

$a = 1 \quad b = 2 \quad d = 0$

$a \quad b^d$

$1 \quad 2^0$

$1 = 1$

$a = b^d$

$T(n) = O(\log n)$.

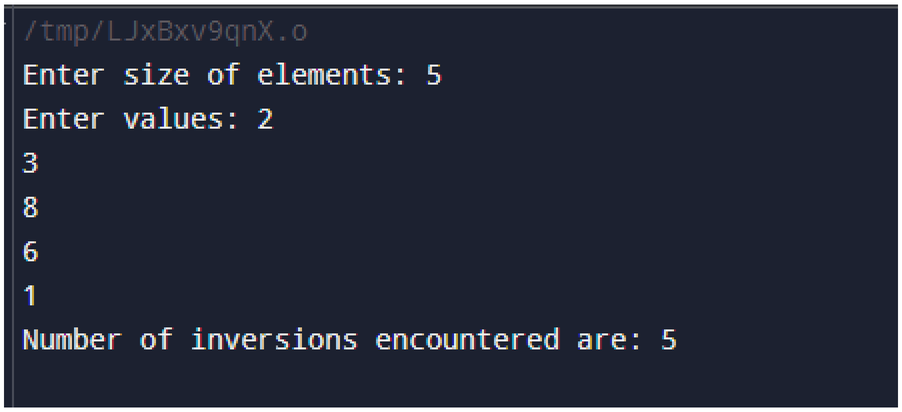
3: The brute force approach involves multiplying a by itself n-1 times, resulting in a time complexity of O(n) for the multiplication operations. In contrast, the divide and conquer technique optimizes this process by decreasing the number of multiplications needed to O(log n), which significantly enhances efficiency for larger values of n.

PART C:

CODE:

```
#include <iostream>

using namespace std;

int inversion_counter(int arr[], int left, int mid, int right) {

    int i = left;

    int j = mid + 1;

    int k = 0;

    int inversion_count = 0;


    int temp[right - left + 1];


    while (i <= mid && j <= right) {

        if (arr[i] <= arr[j]) {

            temp[k++] = arr[i++];

        } else {

            temp[k++] = arr[j++];

            inversion_count += (mid - i + 1);  // Count inversions

        }

    }

    while (i <= mid) {

        temp[k++] = arr[i++];

    }


    while (j <= right) {

        temp[k++] = arr[j++];

    }
    for (int i = 0; i < k; i++) {
        arr[left + i] = temp[i];
    }
    return inversion_count;
}

int merging_count(int arr[], int left, int right) {
    int inversion_count = 0;
    if (left < right) {
        int mid = left + (right - left) / 2;
        inversion_count += merging_count(arr, left, mid);
        inversion_count += merging_count(arr, mid + 1, right);
        inversion_count += inversion_counter(arr, left, mid, right);
    }
    return inversion_count;
}
int main() {
    int n;
    cout << "Enter size of elements: ";
    cin >> n;
    int arr[n];
    cout << "Enter values: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    int result = merging_count(arr, 0, n - 1);
    cout << "Number of inversions encountered are: " << result <<
endl;
}
```

OUTPUT:



```
/tmp/LJxBxv9qnX.o
Enter size of elements: 5
Enter values: 2
3
8
6
1
Number of inversions encountered are: 5
```

The algorithm splits the array into two halves O(log n) and then counts the inversions during merging (O(n) comparisons per merge). Therefore, the overall complexity is O(n log n).

PART D:

Best-Case Complexity:

- O(n log n)
- The best case occurs when the pivot chosen at each step divides the array into two equal halves.
- In this case, the recursion tree is balanced, leading to the logarithmic depth of the recursion tree with each level processing n elements.

Worst-Case Complexity:

- O(n^2)
- The worst case occurs when the pivot chosen is either the smallest or largest element, leading to highly unbalanced partitions.
- In this scenario, one partition contains n−1 elements, while the other contains 0 elements.
- This results in a linear depth recursion tree with n levels, each processing n elements.

PART E:

Logic:

- Sorting the Array: First, we need to sort the array in non-decreasing order. We can use a sorting algorithm like Merge Sort, which has a time complexity of O(n logn).
- Divide Phase: We divide the sorted array into two halves recursively, following the standard approach in the divide-and-conquer method.
- Conquer Phase: Once the array is sorted, the closest pair of elements can be found by checking the differences between consecutive elements. This works because the smallest difference will always be between two adjacent elements.
- Combine Phase: Finally, after finding the closest pairs in each part of the array, we merge these results to find the overall closest pair.

Psuedocode:

```
def closest_pair_1D(arr):
    arr.sort()  //sorting array
    min_diff = float('inf')
    closest_pair = None
    for i in range(1, len(arr)):  // to find closest pair
        diff = abs(arr[i] - arr[i-1])
        if diff < min_diff:
            min_diff = diff
            closest_pair = (arr[i-1], arr[i])
    return closest_pair, min_diff
```

Efficiency

Sorting the array takes O(n logn), and finding the closest pair after sorting takes O(n). Thus, the overall time complexity is O(n logn), making it more efficient than the brute-force approach, which has a complexity of O(n^2).

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

void closest_pair_1D(int arr[], int n, int* first, int* second, int* min_diff) {
    qsort(arr, n, sizeof(int), compare);
    *min_diff = INT_MAX;

    for (int i = 1; i < n; i++) {
        int diff = abs(arr[i] - arr[i-1]);
        if (diff < *min_diff) {
            *min_diff = diff;
            *first = arr[i-1];
            *second = arr[i];
        }
    }
}

int main() {
    int arr[] = {2, 1, 1, 7, 8};
    int n = sizeof(arr) / sizeof(arr[0]);
    int first, second, min_diff;

    closest_pair_1D(arr, n, &first, &second, &min_diff);

    printf("Closest Pair: (%d, %d) with difference: %d\n", first, second, min_diff);

    return 0;
}
```

IS IT BETTER?

- Efficiency:
    - This algorithm is faster than just comparing every possible pair (brute force), which has a time complexity of O(n^2). By sorting the array first, we reduce the problem to simply checking adjacent elements, making it much quicker overall, with a time complexity of O(n logn).

- Accuracy:
  - Sorting the array helps ensure that the smallest difference between numbers is always found between consecutive elements. This means the algorithm will reliably identify the closest pair of numbers.
- Scalability:
  - Because it's more efficient, this approach works well even when the array has many elements. It can handle larger datasets without getting too slow, making it a good choice for bigger arrays.
- **NOTE:**
  - Potential Downsides:
    - The sorting step might use extra memory for large arrays, which could slow things down a bit. But overall, it's still a solid, effective, and accurate algorithm for finding the closest pair in a 1D array.

PART F

```
#include <stdio.h>

int findPeak(int arr[], int low, int high, int n) {

    while (low <= high) {

        int mid = low + (high - low) / 2;

        int left, right;

        if (mid > 0) {

            left = arr[mid - 1];

        } else {

            left = -1;

        }

        if (mid < n - 1) {

            right = arr[mid + 1];

        } else {

            right = -1;

        }

        if (left <= arr[mid] && right <= arr[mid]) {

            return mid;

        }

        if (left > arr[mid]) {

            high = mid - 1;

        } else {

            low = mid + 1;

        }

    }

    return -1;
}


int main() {

    int arr[] = {1, 3, 8, 12, 9, 4, 2};

    int n = sizeof(arr) / sizeof(arr[0]);

    int peakIndex = findPeak(arr, 0, n - 1, n);


    if (peakIndex != -1) {

        printf("Peak with value %d, found at index: %d\n",
arr[peakIndex], peakIndex);

    } else {

        printf("No peak found\n");

    }

}
```

Part G

The given problem aims to find the maximum profit from buying and selling stock over n days using a *divide-and-conquer strategy* with O(n logn) complexity. The approach begins by *dividing the array of stock prices* into two halves: the first half S and the second half S'. This division simplifies the problem by allowing it to be tackled recursively in smaller sections. The algorithm first calculates the maximum profit that can be achieved entirely within the first half S, as well as within the second half S', addressing each part separately.

However, to ensure that the best profit is found, the algorithm must also consider a *cross-boundary solution*, where buying occurs in the left half S and selling happens in the right half S'. The overall solution is derived by taking the *maximum* of three possibilities: the profit from the left half, the profit from the right half, and the cross-boundary profit. This approach ensures that the maximum profit is identified while maintaining the O(n logn) efficiency, as each recursive step involves merging results across the divided halves.

Code:

```
#include <iostream>

#include <vector>

using namespace std;


pair<int, pair<int, int>> maxCrossingProfit(const vector<int>& prices, int
left, int mid, int right) {

    int minPriceLeft = prices[left];

    int minIndexLeft = left;

    for (int i = left; i <= mid; ++i) {

        if (prices[i] < minPriceLeft) {

            minPriceLeft = prices[i];

            minIndexLeft = i;

        }

    }


    int maxPriceRight = prices[mid + 1];

    int maxIndexRight = mid + 1;

    for (int i = mid + 1; i <= right; ++i) {
        if (prices[i] > maxPriceRight) {

            maxPriceRight = prices[i];

            maxIndexRight = i;

        }

    }


    int maxProfit = maxPriceRight - minPriceLeft;

    return {maxProfit, {minIndexLeft, maxIndexRight}};

}


pair<int, pair<int, int>> maxProfitHelper(const vector<int>& prices, int
left, int right) {

    if (left == right) {

        return {0, {left, right}};

    }


    int mid = (left + right) / 2;


    auto leftResult = maxProfitHelper(prices, left, mid);

    int leftProfit = leftResult.first;

    int leftBuy = leftResult.second.first;

    int leftSell = leftResult.second.second;


    auto rightResult = maxProfitHelper(prices, mid + 1, right);

    int rightProfit = rightResult.first;

    int rightBuy = rightResult.second.first;

    int rightSell = rightResult.second.second;


    auto crossResult = maxCrossingProfit(prices, left, mid, right);

    int crossProfit = crossResult.first;

    int crossBuy = crossResult.second.first;

    int crossSell = crossResult.second.second;
    if (leftProfit >= rightProfit && leftProfit >= crossProfit) {
        return {leftProfit, {leftBuy, leftSell}};
    } else if (rightProfit >= leftProfit && rightProfit >= crossProfit) {
        return {rightProfit, {rightBuy, rightSell}};
    } else {
        return {crossProfit, {crossBuy, crossSell}};
    }
}

void maxProfit(const vector<int>& prices) {
    int n = prices.size();
    auto result = maxProfitHelper(prices, 0, n - 1);
    int profit = result.first;
    int buyDay = result.second.first;
    int sellDay = result.second.second;

    if (profit > 0) {
        cout << "Buy on day " << buyDay + 1 << ", Sell on day " << sellDay +
1 << ", Max Profit: " << profit << endl;
    } else {
        cout << "No profit can be made." << endl;
    }
}

int main() {
    vector<int> prices = {9, 1, 5, 3, 7, 8, 4, 2, 10, 6};
    maxProfit(prices);
    return 0;
}
```

PART H

i) To find the median from two sorted databases, each with n distinct elements, we can use a binary search approach that runs in O(log n) time. The task is to identify the nth smallest element among the combined 2n elements. By defining two pointers (low and high) for the range of indices in DB1, we use binary search to adjust these pointers. For each midpoint in DB1, we calculate a corresponding index in DB2 that partitions the arrays around the median. Depending on whether the partition is valid, we either shift the search to the right or left until the median is found.

The binary search ensures that the median is obtained efficiently, using at most O(logn) queries. We perform comparisons between the elements at the partition indices of both databases to maintain the partition conditions, ultimately returning the nth smallest element as the median. The approach reduces the search space by half with each iteration, making it optimal for the problem.

Code:

```
#include <iostream>
using namespace std;

int getKth(int DB1[], int DB2[], int n, int k) {
    int low = max(0, k - n), high = min(k, n);
    while (low < high) {
        int mid1 = (low + high) / 2;
        int mid2 = k - mid1;
        int val1 = (mid1 < n) ? DB1[mid1] : DB2[n-1] + 1;
        int val2 = (mid2 > 0) ? DB2[mid2 - 1] : DB1[0] - 1;
        if (val1 < val2) {
            low = mid1 + 1;
        } else {
            high = mid1;
        }
    }
    int mid1 = low, mid2 = k - low;
    int leftMax1 = (mid1 > 0) ? DB1[mid1 - 1] : DB2[0] - 1;
    int leftMax2 = (mid2 > 0) ? DB2[mid2 - 1] : DB1[0] - 1;

    return (leftMax1 > leftMax2) ? leftMax1 : leftMax2;
}

int findMedian(int DB1[], int DB2[], int n) {
    return getKth(DB1, DB2, n, n);
}

int main() {
    int DB1[] = {1, 5, 9, 11, 15};
    int DB2[] = {4, 2, 8, 12, 16};
    int n = 5;

    int median = findMedian(DB1, DB2, n);
    cout << "Median is: " << median << endl;

    return 0;
}
```

The provided code finds the median of two sorted arrays, DB1 and DB2, each containing n elements. The code achieves this using a binary search approach with an O(logn) time complexity. It leverages the idea that the median of two sorted arrays can be determined by finding the nth smallest element across the combined 2n elements. The function getKth performs a binary search on DB1 while dynamically calculating a corresponding index in DB2 to maintain balance between the two arrays. At each step, it checks whether the potential median conditions are met by comparing the selected elements from both arrays. The function adjusts the search range (low and high) based on these comparisons. Edge cases are handled by setting appropriate boundaries for the values considered, ensuring that the conditions of the merged arrays are maintained throughout the search. Once the correct partitioning is found, the function returns the maximum of the left partition, which corresponds to the median. The findMedian function calls getKth with k = n to directly compute the median as the nth smallest element.

II)

To count the number of significant inversions—efficiently in O(n logn) time, we can use a modified merge sort. The idea is to split the array into two halves recursively, count the significant inversions within each half, and then count significant inversions across the two halves during the merging step. This approach works because when merging, the left and right halves are already sorted, making it easy to identify significant inversions by using a two-pointer technique. For each element in the left half, we find how many elements in the right half are less than half of the current element, indicating a significant inversion.

The merge step is key to maintaining the O(nlogn) complexity. During the merge, as we compare elements from the two halves, we keep track of how many elements in the right half are less than half of the current element from the left half. This counting, combined with the recursive division of the array, ensures that each level of recursion operates in linear time, resulting in an overall O(nlogn) time complexity.

Code:

```
#include <iostream>

#include <vector>

using namespace std;


int mergeAndCount(vector<int>& arr, int left, int mid, int right) {

    int count = 0, j = mid + 1;


    for (int i = left; i <= mid; i++) {

        while (j <= right && arr[i] > 2LL * arr[j]) {

            j++;

        }

        count += (j - (mid + 1));

    }


    vector<int> temp;

    int i = left, k = mid + 1;
    while (i <= mid && k <= right) {
        if (arr[i] <= arr[k]) {
            temp.push_back(arr[i++]);
        } else {
            temp.push_back(arr[k++]);
        }
    }
    while (i <= mid) temp.push_back(arr[i++]);
    while (k <= right) temp.push_back(arr[k++]);

    for (int i = left; i <= right; i++) {
        arr[i] = temp[i - left];
    }
    return count;
}

int countSignificantInversions(vector<int>& arr, int left, int right)
{
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    int count = countSignificantInversions(arr, left, mid);
    count += countSignificantInversions(arr, mid + 1, right);
    count += mergeAndCount(arr, left, mid, right);

    return count;
}

int main() {
    vector<int> arr = {1, 3, 8, 10, 6, 5};
    int n = arr.size();

    int result = countSignificantInversions(arr, 0, n - 1);
    cout << "Number of significant inversions: " << result << endl;
}
```

The algorithm divides the array into two halves recursively until each half has only one element, similar to the merge sort technique. During the merge step, a two-pointer approach is used to count how many elements in the right half are less than half of each element in the left half, identifying significant inversions across the two halves. After counting these inversions, the two halves are merged back into a sorted order to maintain the sequence's integrity. This approach ensures that the counting and merging steps are performed efficiently, achieving an overall time complexity of O(n logn).

III)

Consider an n-node complete binary tree T, where n = 2^d − 1 for some d. Each node v of T is labeled with a real number xv. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a local minimum if the label xv is less than the label xw for all nodes w that are joined to v by an edge. You are given such a complete binary tree T, but the labeling is only specified in the following implicit way: for each node v, you can determine the value xv by probing the node v. Show how to find a local minimum of T using only O(log n) probes to the nodes of T.

CODE:

```
#include <iostream>
#include <vector>
using namespace std;

int getLabel(const vector<int>& labels, int index) {
    return labels[index];
}

int findLocalMin(const vector<int>& labels, int index, int n) {
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    int current = getLabel(labels, index);
    int leftVal = (left < n) ? getLabel(labels, left) : INT_MAX;
    int rightVal = (right < n) ? getLabel(labels, right) : INT_MAX;

    if (current < leftVal && current < rightVal) {
        return index;
    } else if (leftVal < rightVal) {
        return findLocalMin(labels, left, n);
    } else {
        return findLocalMin(labels, right, n);
    }
}

int main() {
    vector<int> labels = {10, 15, 20, 30, 25, 18, 22};
    int n = labels.size();

    int localMinIndex = findLocalMin(labels, 0, n);
    cout << "Local minimum is at index: " << localMinIndex
         << " with value: " << labels[localMinIndex] << endl;

    return 0;
}
```

The search for a local minimum starts at the root node of the binary tree and proceeds recursively. At each node, the algorithm compares the node's value with the values of its left and right children. If the current node's value is smaller than both children's values, it is identified as a local minimum, and the search stops. If not, the algorithm continues the search by moving to the child with the smaller value, indicating a potential local minimum in that direction. Since only one side of the tree is explored at each step, the number of probes is proportional to the tree's height. This results in an efficient solution with O(logn) time complexity.