VISHAL KUMAR **221201** **QUESTION-1**

Part(a):- Can obtain the Final wealth of each node by using recurrance relation followed by matrix exponentiaion technique-

The Pseudo code for the following is below-

```
int m;//No of months
struct node{
    int data;
    node* left;
    node* right;
    node(int val)
    {
        data = val;
        left = NULL;
        right = NULL;
    }
};
// Function to create a BST
node* insert(node* root, int val) {
    if (root == nullptr)
        return new TreeNode(val);

    if (val < root->data)
        root->left = insert(root->left, val);
    else if (val > root->data)
        root->right = insert(root->right, val);

    return root;
}
Int n;
int initial_wealth_arr[n];

// Function to perform BFS traversal and store values in the desired order using
arrays
void storeTreeValuesInOrder(node* root, int* result, int n) {
    if (root == nullptr)
        return;

    node* queue[n]; // Custom queue using an array
    int front = 0, rear = 0;

    queue[rear++] = root;

    int index = 0;
```

VISHAL KUMAR **221201** **QUESTION-1**

```
    while (front < rear && index < n) {
        node* current = queue[front++];
        initial_wealth_arr[index++] = current->data;

        if (current->left)
            queue[rear++] = current->left;

        if (current->right)
            queue[rear++] = current->right;
    }
}
for(i=0 to n)
for(j=0 to n)
{
if( i==j )
{
if(i< (pow(2,n-1)-1))
mat[i][j] = 1/2
else
mat[i][j] = 1;
}
else if(i==(2*j+1))
{
    mat[i][j] = 1/4;
}
else
mat[i][j]=0
}

Matrix_Multiply(a[][n],b[][n])
{
int c[n][n] ={{0}};
    for(i=0 to n)
    for(j=0 to n)
    for(k=0 to n)
    {
        c[i][j] += b[i][k]*c[k][j];
    }

    for(i=0 to n)
    for(j=0 to n)
    {
        a[i][j] = c[i][j];
    }
}
```

VISHAL KUMAR **221201** **QUESTION-1**

```
Matrix_exponentiation(matrix[][n],x)
{
    if(x<=1)return;
    Matrix_exponentiation(matrix, x / 2);
    MatrixMultiply(matrix, matrix);

    if (x % 2 != 0)
        Matrix_Multiplication(matrix, mat, k);//Passing the originally created
"mat" matrix

}

Final_result(matrix[][n])
{

  int  final_wealth_arr[n];//arr willl conatain final wealths of each node taken
in same order as in initial_wealth_arr[n]


    //Call Matrix_exponentiation function with arguments as original matrix mat
created at top and months/12
    Matrix_exponentiation(mat,m/12);

    //After the above operation mat will be mat to the power m/12
    for(i=0 to n)
    for(i=0 to n)
   {
    mat[i][j] = mat[i][j]*pow(2,m);
   }


    for(i=0 to n)
    {
    final_wealth_arr[i] = 0;
    for(j=0 to n)
    {
        final_wealth_arr[i] = final_wealth_arr[i] +
initial_wealth_arr[j]*mat[i][j];
    }

    }
    //Now the final_wealth_arr will store the final wealths of each node in the
order defined

}
```

VISHAL KUMAR **221201** **QUESTION-1**

Part(b):-

1. **Initialization of mat Matrix:**
    - The code initializes the **mat** matrix in a nested loop where **i** and **j** both iterate from 0 to **n**. This initialization has a time complexity of O(n^2).

2. **Insert** Function:

- Time Complexity: O(log N) in the average case, where N is the number of nodes in the BST.
- Explanation: In a balanced BST, the height of the tree is logarithmic in the number of nodes. Therefore, the average case time complexity for inserting a node is O(log N).

3. **StoreTreeValuesInOrder** Function:

- Time Complexity: O(n), where n is the number of nodes in the BST.
- Explanation: The **storeTreeValuesInOrder** function performs a breadth-first traversal of the BST using a custom queue. It visits each node once, so the time complexity is O(n) since it processes all N nodes.

4. **Matrix Multiplication (Matrix_Multiply function):**
    - The **Matrix_Multiply** function performs matrix multiplication using three nested loops, where **i**, **j**, and **k** each iterate from 0 to **n**. So, the time complexity for this operation is O(n^3).

5.Matrix Exponentiation (Matrix_exponentiation function):

- This function recursively divides **x** by 2 until **x** becomes less than or equal to 1, performing matrix multiplication at each step. The matrix multiplication operation (**Matrix_Multiply**) has a time complexity of O(n^3), and the number of recursive calls is determined by the value of **x**.

- The number of recursive calls is roughly O(log(x)) because **x** is halved at each step. Each recursive call involves matrix multiplication with a matrix of size **n x n**.
- Therefore, the overall time complexity of the **Matrix_exponentiation** function is O(n^3*log(x)).

6.Note that we assume that power function takes log(n) time.

7.In the function **Final_result** the overall time complexity is - n+log(n)+(log(m)+1)*n^2+ n^3*(log(m)) which is equal to O(n^3*log(m)).

1. **Correctness Proof**:
   - The correctness of this algorithm can be proven through induction and the properties of matrix exponentiation.
   - Base Case (1 month):
     - After 1 month, each company's wealth doubles. This corresponds to multiplying the initial wealth distribution vector by 2.
     - This is accurately represented by the algorithm.
   - Inductive Hypothesis:
     - Assume the algorithm accurately calculates the wealth distribution after **k** months for some integer **k**.
   - Inductive Step (k+1 months):
     - After **k+1** months, the wealth doubles (**k** months of doubling) and half is distributed to the children at the end of the year.
     - By the induction hypothesis, we have correctly computed the wealth distribution after **k** months.
     - Multiplying **mat** (representing the wealth distribution after **k** months) by itself (**mat**^2) represents wealth distribution after **2k** months (doubling).

- Scaling **mat**^2 by 2 (doubling) corresponds to wealth distribution after **2k + 1** months.
- The final year-end distribution operation (half is distributed to children) is correctly accounted for in the algorithm.

- By induction, the algorithm correctly calculates the wealth distribution after **m** months.

2. **Complexity Analysis**:
   - As previously mentioned, the time complexity of the algorithm is dominated by the **Matrix_exponentiation** function, which is O(log(m) * n^3), where **N** is the number of companies and **m** is the number of months.

In conclusion, the algorithm is correct because it accurately models the wealth distribution process described in the problem statement and uses matrix exponentiation to efficiently calculate the final wealth distribution after **m** months. The correctness is proven by induction, and the time complexity is analyzed.

Part(c):-If there is only one node then final wealth of the company after m months is given by following function

```
int single_node(int wealth,int months)
{
    int p = 1LL<<months;
    return p*wealth;
}
```

The function uses left shift operator to calculate the result in O(1) time.

VISHAL KUMAR **221201** **QUESTION-02**

Part(a):- Pseudo Code, Time Complexity – O(n)

```
int minCost(int a[],int n,int p,int cost_per_room)
{
    int left=0;
    int right = 0;
    int sum =0;
    int minLength = n+1;
    while(right<n)
    {
        sum += a[right];
        while(sum>=p)
        {
            minLength= min(minLength,right-left+1);
            sum -= a[left];
            left++;
        }
        right++;

    }
    if(minLength == n+1)return 0;
    else return minLength*cost_per_room;
}
```

Here in the above code,I am finding the minimum length of the subarray whose sum is >=p .Since the cost of each room is same so we return cost_per_room*minLength ,that is the minimum total cost. If p>total sum of the array then ,return 0.

1. Initialize pointers **left** and **right** to 0, **sum** to 0, and **minLength** to **n+1**.
2. Iterate while **right** is within bounds:
    - Add **a[right]** to **sum**.
    - While **sum** is greater than or equal to **p**:
        - Update **minLength** with the current length.
        - Subtract **a[left]** from **sum** and move **left** rightward.
    - Move **right** rightward.
3. If **minLength** remains unchanged, return 0; otherwise, return **minLength * cost_per_room.**

VISHAL KUMAR **221201** **QUESTION-02**

Part(b):-

```
// Function to find the subarray of maximum length with GCD >= k
int B[MAXN][20]; // Assuming log2(MAXN) is not greater than 20

int findMaxSubarrayWithGCD(int A[], int n, int k) {
    // Create the B matrix
    int logn = log2(n) + 1;
    for (int i = 0; i < n; ++i) {
        B[i][0] = A[i];
    }

    for (int j = 1; (1 << j) <= n; ++j) {
        for (int i = 0; i + (1 << j) <= n; ++i) {
            B[i][j] = gcd(B[i][j - 1], B[i + (1 << (j - 1))][j - 1]);
        }
    }
    // Initialize the result
    int maxLen = 0;

    // Apply two-pointer approach to find the maximum subarray
    for (int left = 0; left < n; ++left) {
        int right = left;
        int currentGCD = A[left];

        while (right < n && currentGCD >= k) {
            int j = log2(right - left + 1); // Find the largest power of 2
            int newGCD = gcd(currentGCD, B[left][j]);

            if (newGCD >= k) {
                maxLen = max(maxLen, right - left + 1);
                // Move the left pointer to the right
                currentGCD = newGCD;
                ++left;
            } else {
                // Expand the subarray to the right
                ++right;
                if (right < n) {
                    currentGCD = gcd(currentGCD, A[right]);
                }
            }
        }
    }
```

```
    return maxLen;
}
```

1. **B** is a 2D array that will be used to store precomputed GCD values for subarrays of the input array **A**.
2. Define the **findMaxSubarrayWithGCD** function:

This function takes an array A, its size n, and an integer k as input and returns the length of the maximum subarray with a GCD greater than or equal to k.

Initialize the B matrix and calculate its values:

1. Here, the code initializes the **B** matrix by calculating the GCD of subarrays. It precomputes GCD values to optimize future GCD calculations.
2. Initialize variables and perform two-pointer approach:

The section initializes variables to keep track of the maximum subarray length (**maxLen**), and it applies a two-pointer approach to find the maximum subarray with a GCD greater than or equal to **k**. It uses the precomputed GCD values from the **B** matrix to optimize GCD calculations.

Part(c):-

Proof of correctness for part(a):-

Assertion: While 0<=right<=n.There exists a minimum length subarray of length **right - left + 1** whose sum of elements is greater than or equal to **p**.

Proof:

1. **Initialization/Base Case:** Before entering the **while** loop, **left** and **right** are both initialized to 0. At this point, the subarray contains only the element at index 0, and **sum** is equal to **a[0]**. If **a[0] >= p**, then this subarray of length 1 satisfies the assertion. Otherwise, if **a[0] < p**, we need to expand the subarray to satisfy the assertion.
2. Maintenance: In each iteration of the outer while loop, we add the element a[right] to the subarray, effectively extending its length by one. We then enter the inner while loop to adjust the subarray boundaries.

3. a. Inner While Loop: We keep moving the left pointer to the right (shrinking the subarray) as long as the current subarray sum sum is greater than or equal to p. This ensures that we find the minimum length subarray with a sum greater than or equal to p.
4. b. Updating minLength: After exiting the inner while loop, we update minLength to be right - left + 1 if the current subarray length is smaller than the previous value of minLength. This ensures that minLength represents the minimum length of a subarray ending at right with a sum greater than or equal to p.
5. Termination: The outer while loop continues until right reaches n, and at each step, we maintain the invariant that minLength represents the minimum length of a subarray ending at right with a sum greater than or equal to p.
6. Final Result: After exiting the while loop, if minLength is still equal to n + 1, it means that no subarray satisfying the condition was found, and the function returns 0, which is correct. Otherwise, it returns minLength, which is the minimum length of a subarray with a sum greater than or equal to p, as asserted.

Here's the reasoning behind the O(n) time complexity for part(a):

1. The **left** and **right** pointers each traverse the array **a**[] at most once.
2. For each iteration of the outer loop, the **right** pointer moves one step to the right, and for each iteration of the inner loop, the **left** pointer also moves at most one step to the right.
3. Since both **left** and **right** pointers traverse the array only once, the total number of iterations of both loops combined is at most 2n, which is still linear with respect to the size of the input array.
4. Also T(n) = T(n-1) + O(1)

VISHAL KUMAR **221201** **QUESTION-03**

```
Part(a)
struct node {
    int data;
    node* left;
    node* right;
    node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

void find_swapped_nodes(node* root, node*& prev, node*& first, node*& second) {
    if (root == NULL)
        return;

    find_swapped_nodes(root->left, prev, first, second);

    if (prev != NULL && root->data < prev->data) {
        if (first == NULL) {
            first = prev;
        }
        second = root;
    }

    prev = root;

    find_swapped_nodes(root->right, prev, first, second);
}

node* find_common_ancestor(node* root, node* node1, node* node2) {
    if (root == NULL)
        return NULL;

    if (root->data > node1->data && root->data > node2->data) {
        return find_common_ancestor(root->left, node1, node2);
    } else if (root->data < node1->data && root->data < node2->data) {
        return find_common_ancestor(root->right, node1, node2);
    }

    return root;
}
```

1. **Structure Definition (struct node):**
    - Defines a structure named **node** representing a node in a BST.

- Each node contains an integer value (**data**), and pointers to its left and right children (**left** and **right**).

2. **Finding Swapped Nodes (find_swapped_nodes function):**
   - Recursively traverses the BST in-order (left-root-right).
   - Compares the current node's value with the previous node's value to identify swaps.
   - Keeps track of the first and second swapped nodes (**first** and **second**) if detected.
3. **Finding Common Ancestor (find_common_ancestor function):**
   - Recursively finds the common ancestor of two given nodes within the BST.
   - Follows the properties of a BST to determine the path to the common ancestor.

Part(b):-

```
struct node {
    int data;
    node* left;
    node* right;
    node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

// Node structure to store information about the nodes
struct NodeInfo {
    int value;
    int position_in_original_BST;
};

// Helper function for in-order traversal and collecting node information
int inOrderTraversal(node* root, NodeInfo nodes_info[], int position) {
    if (!root) return position;

    position = inOrderTraversal(root->left, nodes_info, position);
```

```
    // Store information about the node
    nodes_info[position].value = root->data;
    nodes_info[position].position_in_original_BST = position;
    position++;

    position = inOrderTraversal(root->right, nodes_info, position);

    return position;
}

// Function to find rearranged nodes and k
void findRearrangedNodes(node* root, int G, int& k, int*&
rearranged_node_indices) {
    int n = 0; // Total nodes in the BST
    inOrderTraversal(root, nullptr, n);

    NodeInfo nodes_info[n];
    int position = 0;
    inOrderTraversal(root, nodes_info, position);

    sort(nodes_info, nodes_info + n, [](const NodeInfo& a, const NodeInfo& b) {
        return a.value < b.value;
    });

    k = 0;
    rearranged_node_indices = new int[n];

    for (int i = 0; i < n; i++) {
        if (nodes_info[i].position_in_original_BST != i) {
            rearranged_node_indices[k] = i;
            k++;
        }
    }
}
```

1. **struct node**: Defines a structure for binary tree nodes with integer data, left child pointer, and right child pointer. The constructor initializes the data and sets the left and right pointers to NULL.
2. **struct NodeInfo**: Defines a structure to store information about nodes in the BST. It contains the node's value and its position in the original BST.
3. **inOrderTraversal**: A recursive helper function that performs an in-order traversal of the binary tree. It collects information about each node (value and position) and returns the updated position.

4. **findRearrangedNodes**: The main function that finds rearranged nodes and calculates 'k'. It starts by determining the total number of nodes in the BST using the in-order traversal. Then, it creates an array of NodeInfo objects to store node information.
    - It performs a second in-order traversal to fill the NodeInfo array.
    - Sorts the NodeInfo array based on the node values.
    - Initializes 'k' to 0 and allocates memory for an array of integers to store rearranged node indices.
    - Iterates through the sorted NodeInfo array to find nodes whose positions have changed from the original BST and records their indices in the rearranged_node_indices array.

The overall time complexity of the given code can be analyzed as follows:

The two in-order traversals of the binary search tree both have a time complexity of O(N), where N is the number of nodes in the BST. This is because each node is visited once.

Sorting the NodeInfo array takes O(N log N) time complexity in the worst case, assuming a comparison-based sorting algorithm like quicksort is used.

The final loop that iterates through the sorted NodeInfo array and counts rearranged nodes has a time complexity of O(N), as it iterates through the array once.

So,the overall time complexity of the code O(N log N).

VISHAL KUMAR **221201** **QUESTION-04**

```
Part(a)
int determine_k(int a[],int n,int target)
{
   int L = 1;
   int R = n;
   int mid =0;
   while(L<=R)
   {
     int mid = (L+R)/2;
     if(mid == target )
     {
        return mid;
     }
     else if(target>mid)
     {
        L = mid+1;
     }
     else{
        R = mid-1;
     }
   }
}
```

The code is using binary search to find the index of the target value within the array. Binary search works by repeatedly dividing the search range in half until the target is found or the search range becomes empty.

In each iteration of the while loop, the code:

1. Calculates the middle index **mid** of the current search range, which takes constant time O(1).
2. Compares the target value with the value at the middle index. If they are equal, it returns the index, which also takes constant time O(1).
3. If the target value is greater than the middle value, it updates the left boundary **L** to **mid + 1**.
4. If the target value is less than the middle value, it updates the right boundary **R** to **mid - 1**.

Part(b)

At each iteration, the size of the search range is halved. This is why the algorithm's time complexity is logarithmic with respect to the size of the array 'n'.

VISHAL KUMAR **221201** **QUESTION-04**

$T(n) = T(n/2) + c;$

In each iteration, the number of remaining possibilities is reduced by half, leading to a time complexity of O(log n).

In summary, because the code reduces the search range by half in each iteration, it exhibits logarithmic behavior in terms of time complexity when searching for a target value in a sorted array.

VISHAL KUMAR **221201** **QUESTION-05**

```
void CountPalindrome(string str, int start, int end)
{
    int n = str.length();
    int count = 0;
    if (str[start] == str[end])
    {
        if (start <= n / 2)
        {
            int L = -1;
            int R = start;
            while (L < R - 1)
            {
                int mid = (L + R) / 2;
                if (oracle(mid, (end + start - mid)))
                {
                    R = mid;
                }
                else
                {
                    L = mid;
                }
            }
            count = count + start - R + 1;
        }
       else
        {
            int L = end;
            int R = n;
            while (L < R - 1)
            {
                int mid = (L + R) / 2;
                if (oracle((start + end - mid), mid))
                {
                    L = mid;
                }
                else
                {
                    R = mid;
                }
            }
            count = count + R - end + 1;
        }
    }
}
```

VISHAL KUMAR **221201** **QUESTION-05**

```
int PalindromicSubstrings(string str)
{
    // assuming str[i] as midpoint for odd, and str[i] & str[i+1] for even
    int n = str.length();
    for (int i = 0; i < n; i++)
    {

        // for odd length palindromes

        CountPalindrome(str, i, i);

        // for even length palindromes
        CountPalindrome(str, i, i + 1);
    }
    return count;
}
```

1. **CountPalindrome** is a helper function that counts palindromic substrings in **str** between two indices **start** and **end**. It uses a binary search approach to find palindromes efficiently.
2. It checks if the characters at **start** and **end** are the same. If they are, it proceeds to count palindromes.
3. If **start** is less than or equal to half of the string length (**n/2**), it searches for palindromes with **start** as the center for odd-length palindromes. It uses binary search to expand the palindrome towards the left and right.
4. If **start** is greater than half of the string length, it searches for palindromes with **end** as the center for odd-length palindromes using a similar binary search approach.
5. In both cases, it increments the **count** variable with the number of palindromes found.
6. **PalindromicSubstrings** is the main function that counts all palindromic substrings in the input string **str**.
7. It iterates through each character of **str** and calls **CountPalindrome** twice: once assuming the current character as the center for odd-length palindromes and once assuming the current character and the next character as the center for even-length palindromes.Finally, it returns the total count of palindromic substrings found.