1)
(a) Postorder and Level-order:
No, it is not possible to uniquely identify a tree using only the postorder and level-order traversals.
Reason:
In a postorder traversal, the root node is visited last, while in a level-order traversal, the root node is visited first. Without additional information, such as an inorder traversal, it is impossible to determine the exact position of nodes in the left and right subtrees.
Pseudocode to construct a tree from postorder and level-order traversals

```
TreeNode* buildTreeFromPostLevel(vector<int>& postorder, vector<int>& levelorder) {
    if (postorder.empty() || levelorder.empty())
        return NULL;

    int rootVal = postorder.back();
    TreeNode* root = new TreeNode(rootVal);

    auto it = find(levelorder.begin(), levelorder.end(), rootVal);
    int index = distance(levelorder.begin(), it);

    vector<int> leftLevel(levelorder.begin(), levelorder.begin() + index);
    vector<int> rightLevel(levelorder.begin() + index + 1, levelorder.end());

    postorder.pop_back();

    root->right = buildTreeFromPostLevel(postorder, rightLevel);
    root->left = buildTreeFromPostLevel(postorder, leftLevel);

    return root;
}
```

(b)Inorder and Preorder:
Yes, it is possible to uniquely identify a tree using inorder and preorder traversals.
Reason:
In a preorder traversal, the root node is visited first, while in an inorder traversal, the root node is visited between the left and right subtrees. This allows us to identify the root node and the structure of the left and right subtrees.

Pseudocode to construct a tree from inorder and preorder traversals

```
TreeNode* buildTreeFromInPre(vector<int>& inorder, vector<int>& preorder, int inStart, int inEnd, int& preIndex) {
    if (inStart > inEnd)
        return NULL;

    int rootVal = preorder[preIndex++];
    TreeNode* root = new TreeNode(rootVal);

    auto it = find(inorder.begin(), inorder.end(), rootVal);
    int index = distance(inorder.begin(), it);

    root->left = buildTreeFromInPre(inorder, preorder, inStart, index - 1, preIndex);
    root->right = buildTreeFromInPre(inorder, preorder, index + 1, inEnd, preIndex);

    return root;
}

TreeNode* buildTreeFromInPre(vector<int>& inorder, vector<int>& preorder) {
    int preIndex = 0;
    return buildTreeFromInPre(inorder, preorder, 0, inorder.size() - 1, preIndex);
}
```

(c) Inorder and Level-order:
No, it is not possible to uniquely identify a tree using only the inorder and level-order traversals.
Reason:
In a level-order traversal, the root node is visited first, followed by its children. However, the position of nodes in the left and right subtrees cannot be uniquely determined without additional information such as preorder or postorder traversal.

Pseudocode to construct a tree from inorder and level-order traversals

```
TreeNode* buildTreeFromInLevel(vector<int>& inorder, vector<int>& levelorder) {
    if (inorder.empty() || levelorder.empty())
        return NULL;

    int rootVal = levelorder.front();
    TreeNode* root = new TreeNode(rootVal);

    auto it = find(inorder.begin(), inorder.end(), rootVal);
    int index = distance(inorder.begin(), it);

    vector<int> leftIn(inorder.begin(), inorder.begin() + index);
    vector<int> rightIn(inorder.begin() + index + 1, inorder.end());

    vector<int> leftLevel, rightLevel;
    for (int val : levelorder) {
        auto pos = find(leftIn.begin(), leftIn.end(), val);
        if (pos != leftIn.end())
            leftLevel.push_back(val);
        else {
            pos = find(rightIn.begin(), rightIn.end(), val);
            if (pos != rightIn.end())
                rightLevel.push_back(val);
        }
    }

    root->left = buildTreeFromInLevel(leftIn, leftLevel);
    root->right = buildTreeFromInLevel(rightIn, rightLevel);

    return root;
}
```

(d) Inorder and Postorder:
Yes, it is possible to uniquely identify a tree using inorder and postorder traversals.
Reason:
In a postorder traversal, the root node is visited last, while in an inorder traversal, the root node is visited between the left and right subtrees. This allows us to identify the root node and the structure of the left and right subtrees.

Pseudocode to construct a tree from inorder and postorder traversals

```
TreeNode* buildTreeFromInPost(vector<int>& inorder, vector<int>& postorder, int inStart, int inEnd, int& postIndex) {
    if (inStart > inEnd)
        return NULL;

    int rootVal = postorder[postIndex--];
    TreeNode* root = new TreeNode(rootVal);

    auto it = find(inorder.begin(), inorder.end(), rootVal);
    int index = distance(inorder.begin(), it);

    root->right = buildTreeFromInPost(inorder, postorder, index + 1, inEnd, postIndex);
    root->left = buildTreeFromInPost(inorder, postorder, inStart, index - 1, postIndex);

    return root;
}

TreeNode* buildTreeFromInPost(vector<int>& inorder, vector<int>& postorder) {
    int postIndex = postorder.size() - 1;
    return buildTreeFromInPost(inorder, postorder, 0, inorder.size() - 1, postIndex);
}
```

(e) Postorder and Preorder:
No, it is not possible to uniquely identify a tree using only the postorder and preorder traversals.
Reason:
Both postorder and preorder traversals start with the root node, but without additional information, it is impossible to determine the exact position of nodes in the left and right subtrees.

2)
The time and space complexity of Burn_Tree_Funct and Burn_Tree is O(n) and O(H) respectively.
The Burn_Tree function burns the binary tree starting from the given target node. It prints the nodes in the subtree rooted at the target node using a modified postorder traversal. Once the target node is found, it prints its key and enqueues its children into a vector. Then, it dequeues each node from the vector, prints its key, and enqueues its children. It continues this process until all nodes in the subtree are printed.

3)
Total time and space complexity is O(n).

Algorithm Description:
First, the program takes the level-order traversal of the binary tree as input.
It constructs a matrix representing the binary tree from the level-order traversal.
It then takes two nodes as input and finds their coordinates (level and position) in the binary tree.
If the two nodes have the same level and their positions divided by 2 are equal, they are siblings; otherwise, they are not.

4)
The average time complexity of the Quicksort algorithm is O(NlogN), where N is the number of task IDs.
The time complexity of selecting a task and removing it is O(1).
Total time complexity is O(nlogn)

Space complexity is O(n) because the program stores the task IDs in a vector.

Algorithm Description:

- The quicksort function sorts the given vector of task IDs using the Quicksort algorithm.
- The Task_Scheduler class initializes with a vector of task IDs, sorts them using Quicksort, and maintains the remaining number of tasks.
- Three methods Select_Yuji_Itadori, Select_Megumi_Fushigoro, and Select_Nobara_Kugisaki are provided to select tasks based on different rules.
- Each selection method selects a task according to the specified rule, updates the remaining number of tasks, and removes the selected task from the list.
- The remove method removes the selected task from the list.
- The size method returns the number of remaining tasks.
- The print_IDs method prints the remaining task IDs.