DS Assignmnet 10 problems

Sanjay P

24mx341

1)Given the root of a binary tree, return the inorder, preorder and postorder traversal of its nodes&#39; values.

Example 1:

Input: root = [1,null,2,3]

Inorder: [1,3,2]

Preorder: [1,2,3]w

Postorder: [3,2,1]

Explanation:

Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

Inorder: [4,2,6,5,7,1,3,9,8]

Preorder: [1,2,4,5,6,7,3,8,9]

Postorder: [4,6,7,5,2,9,8,3,1]

CODE

```
#include<stdio.h>
#include<conio.h>
#include<bool.h>

struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
};
//puthu node onu create panu

struct TreeNode* newNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}

//user kita irunthu binary problem input ha vanganu
struct TreeNode* buildTree() {
    int val;
    char choice;

    printf("Enter node value (-1 for NULL): ");
    scanf("%d", &val);
    if(val==-1)
    {
        return NULL;
    }
        struct TreeNode* root = newNode(val);

        printf("Does node %d have a left child? (y/n): ", val);
    scanf(" %c", &choice);
    if (choice == 'y' || choice == 'Y') {
        root->left = buildTree();

           }

    //root ha return pannu
    return root;
}

int main()
{

    //bianry files ha user kita irunthu vangu
 printf("Build the binary tree:\n");
    struct TreeNode* root = buildTree();
   // Inorder Traversal pannu athukana logic
    struct TreeNode* stack[100];
    int top = -1, inorder[100], inorderIdx = 0;
    struct TreeNode* current = root;
 while (preorderTop != -1) {
        current = preorderStack[preorderTop--];
        if (current != NULL) {
            preorder[preorderIdx++] = current->val;
            preorderStack[++preorderTop] = current->right;
            preorderStack[++preorderTop] = current->left;
        }
    }

 // Postorder Traversal
    struct TreeNode* stack1[100];
    struct TreeNode* stack2[100];
    int stack1Top = -1, stack2Top = -1, postorderIdx = 0;
    int postorder[100];
    stack1[++stack1Top] = root;

        while (stack1Top != -1) {
```

```
        current = stack1[stack1Top--];
        if (current != NULL) {
            stack2[++stack2Top] = current;
            stack1[++stack1Top] = current->left;
            stack1[++stack1Top] = current->right;
        }

    }
    while (stack2Top != -1)
        {
        postorder[postorderIdx++] = stack2[stack2Top--]->val;
    }



    // Print traversals
    printf("Inorder: ");
    for (int i = 0; i < inorderIdx; i++)
        {
        printf("%d ", inorder[i]);
    }

printf("\n");

    printf("Preorder: ");
    for (int i = 0; i < preorderIdx; i++)
        {
        printf("%d ", preorder[i]);
    }
  printf("\n");

    printf("Postorder: ");
    for (int i = 0; i < postorderIdx; i++) {
        printf("%d ", postorder[i]);
    }
    printf("\n");

    return 0;



}
```

DS Assignmnet 10 problems
Sanjay P
24mx341

Problem 2:
Implement a stack using a Linked List which supports the following methods:
int size()
boolean isEmpty()
int top()
void push(int element)
void pop()

CODE:
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Definition for a stack node
struct StackNode {
    int data;
    struct StackNode* next;
};

// Initialize the stack
struct StackNode* top = NULL;
int stackSize = 0;

int main() {
    int choice, value;

    while (1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Top\n");
        printf("4. Size\n");
        printf("5. Is Empty\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Push
                printf("Enter value to push: ");
                scanf("%d", &value);
                {
                    struct StackNode* newNode = (struct StackNode*)malloc(sizeof(struct
StackNode));
```

```
            newNode->data = value;
            newNode->next = top; // Point new node to the current top
            top = newNode; // Update top to the new node
            stackSize++; // Increment size
        }
        printf("Pushed %d onto the stack.\n", value);
        break;

    case 2: // Pop
        if (top == NULL) {
            printf("Stack is empty. Cannot pop.\n");
        } else {
            struct StackNode* temp = top; // Temporary node to hold the top
            top = top->next; // Move top to the next node
            printf("Popped %d from the stack.\n", temp->data);
            free(temp); // Free the popped node
            stackSize--; // Decrement size
        }
        break;

    case 3: // Top
        if (top == NULL) {
            printf("Stack is empty. No top element.\n");
        } else {
            printf("Top element is: %d\n", top->data);
        }
        break;

    case 4: // Size
        printf("Stack size is: %d\n", stackSize);
        break;

    case 5: // Is Empty
        if (top == NULL) {
            printf("Stack is empty.\n");
        } else {
            printf("Stack is not empty.\n");
        }
        break;

    case 6: // Exit
        printf("Exiting...\n");
        // Free the stack before exiting
        while (top != NULL) {
                struct StackNode* temp = top;
                top = top->next;
                free(temp);
            }
            exit(0);

        default:
            printf("Invalid choice. Please try again.\n");
    }
}

    return 0;
}
```

Problem 3:
Implement a queue using a Linked List which supports the following methods:
int size()
boolean isEmpty()
int display()
void enqueue(int element)
void dequeue()

CODE:

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

// Definition for a queue node
struct QueueNode {
    int data;
    struct QueueNode* next;
};

// Initialize the queue
struct QueueNode* front = NULL;
struct QueueNode* rear = NULL;
int queueSize = 0;

int main() {
    int choice, value;

    while (1) {
```

```
    printf("\nQueue Operations:\n");
    printf("1. Enqueue\n");
    printf("2. Dequeue\n");
    printf("3. Display\n");
    printf("4. Size\n");
    printf("5. Is Empty\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
      case 1: // Enqueue
        printf("Enter value to enqueue: ");
        scanf("%d", &value);
        {
          struct QueueNode* newNode = (struct QueueNode*)malloc(sizeof(struct
QueueNode));
          newNode->data = value;
          newNode->next = NULL; // New node will be the last node

          if (rear == NULL) {
            // If the queue is empty, both front and rear will point to the new node
            front = rear = newNode;
          } else {
            // Link the new node at the end of the queue and update rear
            rear->next = newNode;
            rear = newNode;
          }
          queueSize++; // Increment size
        }
        printf("Enqueued %d into the queue.\n", value);
        break;

      case 2: // Dequeue
        if (front == NULL) {
          printf("Queue is empty. Cannot dequeue.\n");
        } else {
          struct QueueNode* temp = front; // Temporary node to hold the front
          front = front->next; // Move front to the next node
          printf("Dequeued %d from the queue.\n", temp->data);
          free(temp); // Free the dequeued node
          queueSize--; // Decrement size

          // If the queue becomes empty, also update rear to NULL
        if (front == NULL) {
            rear = NULL;
        }
    }
    break;

case 3: // Display
    if (front == NULL) {
        printf("Queue is empty.\n");
    } else {
        struct QueueNode* current = front;
        printf("Queue elements: ");
        while (current != NULL) {
            printf("%d ", current->data);
            current = current->next;
        }
        printf("\n");
    }
    break;

case 4: // Size
    printf("Queue size is: %d\n", queueSize);
    break;

case 5: // Is Empty
    if (front == NULL) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue is not empty.\n");
    }
    break;

case 6: // Exit
    printf("Exiting...\n");
    // Free the queue before exiting
    while (front != NULL) {
        struct QueueNode* temp = front;
        front = front->next;
        free(temp);
    }
    exit(0);

default:
    printf("Invalid choice. Please try again.\n");
```

```
        }
    }

    return 0;
}
```

PROBLEM 4:
You are given the root node of a binary search tree (BST) and a value to insert into the tree. Return the root node of the BST after the insertion. It is guaranteed that the new value does not exist in the original BST.
Notice that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return any of them.
Example 1:
Input: root = [4,2,7,1,3], val = 5
Output: [4,2,7,1,3,5]

Example 2:
Input: root = [40,20,60,10,30,50,70], val = 25
Output: [40,20,60,10,30,50,70,null,null,25]
Example 3:
Input: root = [4,2,7,1,3,null,null,null,null,null,null], val = 5
Output: [4,2,7,1,3,5]

CODE:
```
#include <stdio.h>
#include <stdlib.h>

// Definition for a binary tree node
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
};

// Function to create a new tree node
struct TreeNode* newNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

DS Assignmnet 10 problems
Sanjay P
24mx341

```
// Function to display the tree in level order (for testing purposes)
void display(struct TreeNode* root) {
    if (root == NULL) {
        printf("null ");
        return;
    }
    printf("%d ", root->val);
    display(root->left);
    display(root->right);
}

int main() {
    // Manually creating the BST for the example [4,2,7,1,3]
    struct TreeNode* root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(7);
    root->left->left = newNode(1);
    root->left->right = newNode(3);

    int val;
    printf("Enter the value to insert into the BST: ");
    scanf("%d", &val);

    // Insertion logic
    struct TreeNode* current = root;
    struct TreeNode* parent = NULL;

    while (current != NULL) {
        parent = current;
        if (val < current->val) {
            current = current->left;
        } else {
            current = current->right;
        }
    }

    // Create a new node for the value to be inserted
    struct TreeNode* newNodeToInsert = newNode(val);

    // Attach the new node to the parent
    if (val < parent->val) {
        parent->left = newNodeToInsert;
    } else {
        parent->right = newNodeToInsert;
    }

    // Display the tree in level order
    printf("BST after insertion: ");
    display(root);
    printf("\n");

    // Free allocated memory (not shown here for simplicity)

    return 0;
}
```

PROBLEM 5:
You are given the root of a binary search tree (BST) and an integer val. Find the node in the
BST that the node&#39;s value equals val and return the subtree rooted with that node. If such a
node does not exist, return null.
Example 1:

Input: root = [4,2,7,1,3], val = 2
Output: [2,1,3]
Example 2:

Input: root = [4,2,7,1,3], val = 5
Output: []

CODE:
```
#include <stdio.h>
#include <stdlib.h>

// Definition for a binary tree node
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
};

// Function to create a new tree node
struct TreeNode* newNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to display the tree in level order (for testing purposes)
void display(struct TreeNode* root) {
    if (root == NULL) {
        printf("null ");
        return;
    }
    printf("%d ", root->val);
    display(root->left);
    display(root->right);
}

int main() {
    // Manually creating the BST for the example [4,2,7,1,3]
    struct TreeNode* root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(7);
    root->left->left = newNode(1);
    root->left->right = newNode(3);

    int val;
    printf("Enter the value to search in the BST: ");
    scanf("%d", &val);

    // Search logic
    struct TreeNode* current = root;
    struct TreeNode* foundNode = NULL;

    while (current != NULL) {
        if (val == current->val) {
            foundNode = current; // Node found
            break;
        } else if (val < current->val) {
            current = current->left; // Go left
        } else {
            current = current->right; // Go right
        }
    }

    // Display the subtree rooted at the found node
    if (foundNode != NULL) {
        printf("Subtree rooted at node with value %d: ", foundNode->val);
        display(foundNode);
    } else {
        printf("Node with value %d not found in the BST.\n", val);
    }

    // Free allocated memory (not shown here for simplicity)

    return 0;
}
```

PROBLEM 6:
Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.
Basically, the deletion can be divided into two stages:
Search for a node to remove.
If the node is found, delete the node.
Example 1:

Input: root = [5,3,6,2,4,null,7], key = 3
Output: [5,4,6,2,null,null,7]

CODE:

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

// Definition for a binary tree node
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
};

// Function to create a new tree node
struct TreeNode* newNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to display the tree in level order (for testing purposes)
void display(struct TreeNode* root) {
    if (root == NULL) {
        printf("null ");
        return;
    }
    printf("%d ", root->val);
    display(root->left);
    display(root->right);
}

int main() {
    // Manually creating the BST for the example [5,3,6,2,4,null,7]
    struct TreeNode* root = newNode(5);
    root->left = newNode(3);
    root->right = newNode(6);
    root->left->left = newNode(2);
    root->left->right = newNode(4);
    root->right->right = newNode(7);

    int key;
    printf("Enter the key to delete from the BST: ");
    scanf("%d", &key);

    // Deletion logic
    struct TreeNode* current = root;
    struct TreeNode* parent = NULL;

    // Step 1: Find the node to delete
    while (current != NULL && current->val != key) {
        parent = current;
        if (key < current->val) {
            current = current->left;
        } else {
            current = current->right;
        }
    }

    // If the node was not found
    if (current == NULL) {
        printf("Node with key %d not found in the BST.\n", key);
        return 0;
    }

    // Step 2: Delete the node
    // Case 1: Node has no children (leaf node)
```

```
if (current->left == NULL && current->right == NULL) {
    if (current == root) {
        free(current);
        root = NULL; // Tree is now empty
    } else if (parent->left == current) {
        parent->left = NULL;
    } else {
        parent->right = NULL;
    }
}
// Case 2: Node has one child
else if (current->left == NULL || current->right == NULL) {
    struct TreeNode* child = (current->left != NULL) ? current->left : current->right;
    if (current == root) {
        free(current);
        root = child; // Update root
    } else if (parent->left == current) {
        parent->left = child;
    } else {
        parent->right = child;
    }
}
// Case 3: Node has two children
else {
    // Find the inorder successor (smallest in the right subtree)
    struct TreeNode* successor = current->right;
    struct TreeNode* successorParent = current;

    while (successor->left != NULL) {
        successorParent = successor;
        successor = successor->left;
    }

    // Copy the successor's value to the current node
    current->val = successor->val;

    // Delete the successor
    if (successorParent->left == successor) {
        successorParent->left = successor->right;
    } else {
        successorParent->right = successor->right;
    }
    current = successor; // For freeing memory later
}
```

```
    // Free the memory of the deleted node
    free(current);

    // Display the tree after deletion
    printf("BST after deletion: ");
    display(root);
    printf("\n");

    // Free allocated memory (not shown here for simplicity)

    return 0;
}
```

PROBLEM 7:
Given an m x n 2D binary grid ‘grid’ which represents a map of &#39;1&#39;s (land) and &#39;0&#39;s (water),
return the number of islands. Solve using BFS
An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.
Example 1:
Input: grid = [
[&quot;1&quot;,&quot;1&quot;,&quot;1&quot;,&quot;1&quot;,&quot;0&quot;],
[&quot;1&quot;,&quot;1&quot;,&quot;0&quot;,&quot;1&quot;,&quot;0&quot;],
[&quot;1&quot;,&quot;1&quot;,&quot;0&quot;,&quot;0&quot;,&quot;0&quot;],
[&quot;0&quot;,&quot;0&quot;,&quot;0&quot;,&quot;0&quot;,&quot;0&quot;]
]

CODE:
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ROWS 50
#define MAX_COLS 50

int main() {
    // Manually creating the grid for the example
    char grid[MAX_ROWS][MAX_COLS] = {
        {'1', '1', '1', '1', '0'},
        {'1', '1', '0', '1', '0'},
        {'1', '1', '0', '0', '0'},
        {'0', '0', '0', '0', '0'}
    };
```

DS Assignmnet 10 problems
Sanjay P
24mx341

```
int rows = 4; // Number of rows in the grid
int cols = 5; // Number of columns in the grid
int visited[MAX_ROWS][MAX_COLS] = {0}; // To keep track of visited cells
int islandCount = 0;


// Directions for moving up, down, left, right
int directions[4][2] = {
    {-1, 0}, // Up
    {1, 0},  // Down
    {0, -1}, // Left
    {0, 1}   // Right
};

// BFS implementation
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        if (grid[i][j] == '1' && !visited[i][j]) {
            // Found an unvisited land cell, start BFS
            islandCount++;
            visited[i][j] = 1; // Mark as visited

            // BFS queue
            int queue[MAX_ROWS * MAX_COLS][2]; // Queue to hold coordinates
            int front = 0, rear = 0;

            // Enqueue the starting cell
            queue[rear][0] = i;
            queue[rear][1] = j;
            rear++;

            // Process the queue
            while (front < rear) {
                int x = queue[front][0];
                int y = queue[front][1];
                front++;

                // Explore all four directions
                for (int d = 0; d < 4; d++) {
                    int newX = x + directions[d][0];
                    int newY = y + directions[d][1];

                    // Check if the new position is within bounds and is land
                    if (newX >= 0 && newX < rows && newY >= 0 && newY < cols &&
                    grid[newX][newY] == '1' && !visited[newX][newY]) {
                    visited[newX][newY] = 1; // Mark as visited
                    // Enqueue the new position
                    queue[rear][0] = newX;
                    queue[rear][1] = newY;
                    rear++;
                }
            }
        }
      }
    }
  }

  // Output the number of islands
  printf("Number of islands: %d\n", islandCount);

  return 0;
}
```

PROBLEM 8:
Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).
Example 1:

Input: root = [1,2,2,3,4,4,3]
Output: true
Example 2:

Input: root = [1,2,2,null,3,null,3]
Output: false

CODE:
```
#include <stdio.h>
#include <stdlib.h>

// Definition for a binary tree node
struct TreeNode {
  int val;
  struct TreeNode *left;
  struct TreeNode *right;
};

// Function to create a new tree node
```

DS Assignmnet 10 problems
Sanjay P
24mx341

```
struct TreeNode* newNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to display the tree in level order (for testing purposes)
void display(struct TreeNode* root) {
    if (root == NULL) {
        printf("null ");
        return;
    }
    printf("%d ", root->val);
    display(root->left);
    display(root->right);
}

int main() {
    // Manually creating the binary tree for the example [1,2,2,3,4,4,3]
    struct TreeNode* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(4);
    root->right->right = newNode(3);

    // Check if the tree is symmetric
    int isSymmetric = 1; // Assume it is symmetric
    struct TreeNode* left = root->left;
    struct TreeNode* right = root->right;

    // Use a stack to perform an iterative check for symmetry
    struct TreeNode* stack[100]; // Stack for nodes
    int top = -1;

    // Push the left and right nodes onto the stack
    stack[++top] = left;
    stack[++top] = right;

    while (top >= 0) {
        right = stack[top--];
        left = stack[top--];

        // If both nodes are NULL, continue
        if (left == NULL && right == NULL) {
            continue;
        }
        // If one is NULL and the other is not, it's not symmetric
        if (left == NULL || right == NULL) {
            isSymmetric = 0;
            break;
        }
        // If values are not equal, it's not symmetric
        if (left->val != right->val) {
            isSymmetric = 0;
            break;
        }

        // Push children in the order that they will be compared
        stack[++top] = left->left;   // Left child of left node
        stack[++top] = right->right;  // Right child of right node
        stack[++top] = left->right;   // Right child of left node
        stack[++top] = right->left;    // Left child of right node
    }

    // Output the result
    if (isSymmetric) {
        printf("The tree is symmetric.\n");
    } else {
        printf("The tree is not symmetric.\n");
    }

    // Display the tree (optional)
    printf("Tree structure (level order): ");
    display(root);
    printf("\n");

    // Free allocated memory (not shown here for simplicity)

    return 0;
}
```

DS Assignmnet 10 problems
Sanjay P
24mx341

PROBLEM 9
Given the head of a singly linked list, sort the list using insertion sort, and return the sorted list&#39;s head.

The steps of the insertion sort algorithm:
1. Insertion sort iterates, consuming one input element each repetition and growing a sorted output list.
2. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list and inserts it there.
3. It repeats until no input elements remain.
Example 1:

Input: head = [4,2,1,3]
Output: [1,2,3,4]

CODE:
```
#include <stdio.h>
#include <stdlib.h>

// Definition for a singly linked list node
struct ListNode {
    int val;
    struct ListNode *next;
};

// Function to create a new list node
struct ListNode* newNode(int val) {
    struct ListNode* node = (struct ListNode*)malloc(sizeof(struct ListNode));
    node->val = val;
    node->next = NULL;
    return node;
}

// Function to display the linked list
void display(struct ListNode* head) {
    struct ListNode* current = head;
    while (current != NULL) {
        printf("%d -> ", current->val);
        current = current->next;
    }
    printf("NULL\n");
}
```

DS Assignmnet 10 problems
Sanjay P
24mx341

```
int main() {
    // Manually creating the linked list for the example [4,2,1,3]
    struct ListNode* head = newNode(4);
    head->next = newNode(2);
    head->next->next = newNode(1);
    head->next->next->next = newNode(3);

    // Display the original linked list
    printf("Original linked list: ");
    display(head);

    // Insertion sort on the linked list
    struct ListNode* sorted = NULL; // Start with an empty sorted list
    struct ListNode* current = head; // Pointer to traverse the original list

    while (current != NULL) {
        struct ListNode* next = current->next; // Store the next node
        // Insert current into the sorted list
        if (sorted == NULL || sorted->val >= current->val) {
            // Insert at the beginning
            current->next = sorted;
            sorted = current;
        } else {
            // Find the correct position to insert
            struct ListNode* temp = sorted;
            while (temp->next != NULL && temp->next->val < current->val) {
                temp = temp->next;
            }
            // Insert current after temp
            current->next = temp->next;
            temp->next = current;
        }
        current = next; // Move to the next node
    }

    // Display the sorted linked list
    printf("Sorted linked list: ");
    display(sorted);

    // Free allocated memory (not shown here for simplicity)
    // You should free the nodes in both the original and sorted lists

    return 0 ●
```

DS Assignmnet 10 problems
Sanjay P
24mx341

PROBLEM :10
You are given a 2D array of integers envelopes where envelopes[i] = [wi, hi] represents the width and the height of an envelope.
One envelope can fit into another if and only if both the width and height of one envelope are greater than the other envelope&#39;s width and height.
Return the maximum number of envelopes you can Russian doll (i.e., put one inside the other).
Note: You cannot rotate an envelope.
Example 1:
Input: envelopes = [[5,4],[6,4],[6,7],[2,3]]
Output: 3

Explanation: The maximum number of envelopes you can Russian doll is 3 ([2,3] =&gt; [5,4] =&gt;
[6,7]).
Example 2:
Input: envelopes = [[1,1],[1,1],[1,1]]

CODE:
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ENVELOPES 1000

// Comparator function for sorting envelopes
int compare(const void *a, const void *b) {
   int *envelopeA = (int *)a;
   int *envelopeB = (int *)b;
   // First sort by width, then by height in descending order if widths are equal
   if (envelopeA[0] == envelopeB[0]) {
      return envelopeB[1] - envelopeA[1]; // Descending order for height
   }
   return envelopeA[0] - envelopeB[0]; // Ascending order for width
}

int main() {
   int envelopes[MAX_ENVELOPES][2];
   int n;

   // Input the number of envelopes
   printf("Enter the number of envelopes: ");
   scanf("%d", &n);
```

DS Assignmnet 10 problems
Sanjay P
24mx341

```
    // Input the envelopes
    printf("Enter the envelopes (width height):\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &envelopes[i][0], &envelopes[i][1]);
    }

    // Sort the envelopes
    qsort(envelopes, n, sizeof(envelopes[0]), compare);

    // Dynamic programming array to find the longest increasing subsequence based on height
    int dp[MAX_ENVELOPES] = {0};
    int maxEnvelopes = 0;

    // Find the longest increasing subsequence based on height
    for (int i = 0; i < n; i++) {
        dp[i] = 1; // Each envelope can at least contain itself
        for (int j = 0; j < i; j++) {
            if (envelopes[i][1] > envelopes[j][1]) {
                dp[i] = dp[i] > dp[j] + 1 ? dp[i] : dp[j] + 1;
            }
        }
        maxEnvelopes = maxEnvelopes > dp[i] ? maxEnvelopes : dp[i];
    }

    // Output the result
    printf("Maximum number of envelopes you can Russian doll: %d\n", maxEnvelopes);

    return 0;
}
```