CODE 1

```
def knapsack_dynamic_programming(weights, values, capacity):
    """
    Solve the 0-1 Knapsack problem using dynamic programming.

    :param weights: List of weights of the items.
    :param values: List of values of the items.
    :param capacity: Maximum capacity of the knapsack.
    :return: Maximum value that can be obtained, and the items included.
    """
    n = len(values)
    # Create a 2D array to store the maximum value at each n and capacity
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    # Fill dp array
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])
            else:
                dp[i][w] = dp[i - 1][w]

    # Find the items included in the knapsack
    result = dp[n][capacity]
    w = capacity
    items_included = []

    for i in range(n, 0, -1):
        if result <= 0:
            break
        if result == dp[i - 1][w]:
            continue
        else:
            items_included.append(i - 1)
            result -= values[i - 1]
            w -= weights[i - 1]

    return dp[n][capacity], items_included


# Input from the user
num_items = int(input("Enter the number of items: "))
weights = []
```

```
values = []

for i in range(num_items):
    weight = int(input(f"Enter weight of item {i + 1}: "))
    value = int(input(f"Enter value of item {i + 1}: "))
    weights.append(weight)
    values.append(value)

capacity = int(input("Enter the maximum capacity of the knapsack: "))

# Solve the knapsack problem
max_value, items = knapsack_dynamic_programming(weights, values, capacity)
print("Maximum value that can be obtained:", max_value)
print("Items included (0-indexed):", items)
```

CODE 2

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract StudentData {
    // Structure to represent a Student
    struct Student {
        uint256 id;
        string name;
        uint8 age;
        string course;
    }

    // Array to store the list of students
    Student[] public students;

    // Event to log when a student is added
    event StudentAdded(uint256 id, string name, uint8 age, string course);

    // Function to add a new student
    function addStudent(uint256 _id, string memory _name, uint8 _age, string memory _course)
public {
        // Create a new student and push to the array
        students.push(Student(_id, _name, _age, _course));

        // Emit an event when a student is added
        emit StudentAdded(_id, _name, _age, _course);
    }
    // Fallback function
    fallback() external payable {
        revert("Fallback function called. No direct payments allowed.");
    }

    // Function to get the number of students
    function getStudentCount() public view returns (uint256) {
        return students.length;
    }

    // Function to retrieve a student by index
    function getStudent(uint256 index) public view returns (uint256, string memory, uint8, string
memory) {
        require(index < students.length, "Index out of bounds");
        Student memory student = students[index];
        return (student.id, student.name, student.age, student.course);
    }
}
```