1. **Time & Space Complexity of 0/1 Knapsack Algorithm Using Dynamic Programming:**

Time Complexity:

- Let the number of items be n and the maximum weight capacity be W.

- In the dynamic programming approach, we create a 2D table `DP[][]` of size $(n+1) \times (W+1)$, where each entry `DP[i][j]` represents the maximum value that can be achieved using the first i items and a knapsack of weight j.

- For each item, we check two possibilities: either including the item or excluding the item. This results in filling the table by iterating over all items and all possible weights.

- Therefore, the time complexity is proportional to the number of entries in the DP table, which is $O(n \times W)$.

So, the **time complexity** is:

$$
O(n \times W)
$$

Space Complexity:

- The space complexity is dominated by the storage required for the DP table. The table has $n+1$ rows (for each item plus the base case) and $W+1$ columns (for each possible weight from 0 to W)).

- Therefore, the space complexity is:

\[

O(n \times W)

\]

If we want to optimize space, we can use a 1D array to store the results, as we only need the current row and the previous row at any point. In this case, the space complexity would be reduced to \(O(W)\).

2. **Realistic Applications of 0/1 Knapsack Problem:**

Application 1: Budget Allocation

- **Problem**: You are managing a project with a fixed budget. You have a set of tasks to complete, where each task requires a certain amount of money (weight) and has an expected profit or value (value). The goal is to select the tasks such that the total cost does not exceed the budget, while maximizing the total profit.

- **Application**: The 0/1 knapsack algorithm helps in determining which tasks (items) to choose (either include or exclude them) in order to maximize the total profit without exceeding the budget (capacity).

Application 2: Cargo Packing and Shipping

- **Problem**: You are responsible for packing cargo into a truck. Each item has a certain weight and value (such as goods with different costs), and the truck has a weight limit. The goal is to maximize the value of the cargo while respecting the weight limit.

- **Application**: The 0/1 knapsack algorithm can be used to decide which items to load into the truck to maximize the value of the cargo without exceeding the weight limit.

---

3. **0/1 Knapsack Solution for a Weight Limit of 10 Using Dynamic Programming:**

Let's solve a 0/1 knapsack problem with a knapsack capacity of 10 using the dynamic programming approach.

Given Items:

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	8

Knapsack Capacity (W): 10

Steps:

1. **Create a DP Table:**

   - Initialize a 2D DP table where `DP[i][w]` represents the maximum value achievable with the first i items and a knapsack capacity of w.

   - The size of the DP table will be $(4 + 1)$ x $(10 + 1)$ = 5×11.

2. **DP Table Calculation:**

- The DP table is filled based on two choices for each item:

- **Exclude the item**: The value will be the same as the value for the previous item with the same weight limit, i.e., `DP[i-1][w]`.

- **Include the item**: If the item's weight is less than or equal to the current weight capacity, the value will be the item’s value plus the value from the previous row with the remaining capacity, i.e., `DP[i-1][w - weight[i]] + value[i]`.

Filling the Table:

For each item and for each weight from 0 to 10, we compute the maximum value.

Item\Weight	0	1	2	3	4	5	6	7	8	9	10
0 (None)	0	0	0	0	0	0	0	0	0	0	0
1 (2,3)	0	0	3	3	3	3	3	3	3	3	3
2 (3,4)	0	0	3	4	4	4	7	7	7	7	7
3 (4,5)	0	0	3	4	5	5	7	8	8	9	9
4 (5,8)	0	0	3	4	5	8	8	12	13	13	13

Maximum Value: The maximum value that can be obtained with a weight limit of 10 is found in `DP[4][10]` which equals **13**.

Conclusion:

- The optimal selection of items will result in a maximum value of **13**.

- Items selected will be **Item 3 (weight 4, value 5)**, **Item 4 (weight 5, value 8)**, and part of **Item 2 (weight 3, value 4)**.

……code explanation……

Step-by-Step Explanation of the 0/1 Knapsack Dynamic Programming Code:

This program solves the **0/1 Knapsack Problem** using **Dynamic Programming**. It finds the maximum profit that can be obtained with a given knapsack capacity and the weights and profits of different objects. The program also prints the objects selected (1 if taken, 0 if not), as well as the DP table used for solving the problem.

1. Function Declaration:

```cpp
void knapsack(int n, int w, int wt[], int pr[]) {
```

- This function takes four parameters:
  - `n`: The total number of objects.
  - `w`: The capacity of the knapsack.
  - `wt[]`: An array of weights for each item.
  - `pr[]`: An array of profits corresponding to each item.

2. DP Table Initialization:

```cpp
int mat[n+1][w+1];
```

- A 2D array `mat` of size \((n+1) \times (w+1)\) is created.
  - `mat[i][j]` will store the maximum profit achievable with the first `i` items and a knapsack capacity of `j`.
  - We use `n+1` and `w+1` to account for the 0th row and 0th column, which represent the base cases where no items are considered, or the capacity is 0.

3. Populating the DP Table:

```cpp
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= w; j++) {
        if (i == 0 || j == 0) {
            mat[i][j] = 0;
        } else if (wt[i-1] <= j) {
            mat[i][j] = max(pr[i-1] + mat[i-1][j-wt[i-1]], mat[i-1][j]);
        } else {
            mat[i][j] = mat[i-1][j];
        }
    }
}
```

- **Outer loop (`i`)** iterates through all the items (from 0 to `n`).

- **Inner loop (`j`)** iterates through all possible weights (from 0 to `w`).

- **Base Case**:

  - If there are no items or the knapsack has a capacity of 0 (`i == 0` or `j == 0`), the maximum profit is 0, i.e., `mat[i][j] = 0`.

- **Decision**:

  - If the current item can fit in the knapsack (`wt[i-1] <= j`), we choose the maximum of two options:

    - **Include the item**: The profit will be the profit of the item plus the value from the previous row with the remaining capacity (`pr[i-1] + mat[i-1][j-wt[i-1]]`).

    - **Exclude the item**: The profit remains the same as from the previous row (`mat[i-1][j]`).

  - If the item cannot be included, the profit remains the same as the previous row (`mat[i-1][j]`).

4. Output the Maximum Profit:

```cpp
cout << "Maximum profit: " << mat[n][w] << endl;
```

- The maximum profit will be stored in `mat[n][w]`, which is the last element of the DP table, representing the maximum profit achievable with all items and the given knapsack capacity.

5. Backtracking to Find the Selected Items:

```cpp
int remainingCapacity = w;
int taken[n] = {0};

for (int i = n; i > 0; i--) {
   if (mat[i][remainingCapacity] != mat[i-1][remainingCapacity]) {
      taken[i-1] = 1;
      remainingCapacity -= wt[i-1];
   }
}
```

- The array `taken[]` is used to track which items are included in the optimal solution. Initially, all items are marked as 0 (not taken).

- The loop starts from the last item (`i = n`) and works backwards to determine which items were included in the optimal subset.

- If the value at `mat[i][remainingCapacity]` is different from `mat[i-1][remainingCapacity]`, it means that the `i`-th item was included, and we update the `taken[]` array and reduce the remaining capacity by the weight of the item (`remainingCapacity -= wt[i-1]`).

6. Output the Items Selected:

```cpp
cout << "Objects taken (1 = taken, 0 = not taken): ";
cout << "[";
```

```
for (int i = 0; i < n; i++) {
    cout << taken[i];
    if (i < n-1) cout << ", ";
}
cout << "]" << endl;
```

- This prints an array showing which items were selected in the optimal solution. A `1` means the item was included, and a `0` means it was not included.

7. Output the DP Table:

```cpp
cout << "DP table:" << endl;
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= w; j++) {
        cout << mat[i][j] << " ";
    }
    cout << endl;
}
```

- This prints the entire DP table, which helps visualize the process of calculating the optimal solution. Each entry `mat[i][j]` represents the maximum profit for the first `i` items with a knapsack capacity of `j`.

8. Main Function:

```cpp
int main() {
    int n, w;

    cout << "Enter the total number of objects: ";
    cin >> n;

    int wt[n], pr[n];

    for (int i = 0; i < n; i++) {
        cout << "Enter weight for object " << i+1 << ": ";
        cin >> wt[i];
        cout << "Enter profit for object " << i+1 << ": ";
        cin >> pr[i];
    }

    cout << "Enter the capacity: ";
    cin >> w;

    knapsack(n, w, wt, pr);

    return 0;
}
```

```

- **Input**:

  - First, the program asks for the number of objects (`n`) and the knapsack capacity (`w`).

  - Then, for each item, it asks for the weight (`wt[]`) and profit (`pr[]`).

- The `knapsack()` function is called with the provided values for `n`, `w`, `wt[]`, and `pr[]` to compute and display the maximum profit, the selected items, and the DP table.

### **Summary of Program Flow:**

1. Input the number of items, their weights, profits, and the knapsack capacity.

2. Fill the DP table using dynamic programming.

3. Compute the maximum profit.

4. Backtrack through the DP table to determine which items are included in the optimal solution.

5. Output the maximum profit, the items selected, and the DP table.

### **Output Example:**

If you have 4 items with the following weights and profits:

- Weights: 2, 3, 4, 5

- Profits: 3, 4, 5, 8

- Knapsack Capacity: 10
```

The program will display the maximum profit, the items taken, and the DP table.

Example Output:

```
Maximum profit: 12
Objects taken (1 = taken, 0 = not taken): [1, 1, 0, 1]
DP table:
0 0 0 0 0 0 0 0 0 0 0
0 0 3 3 3 3 3 3 3 3
0 0 3 4 4 4 4 7 7 7
0 0 3 4 5 5 7 8 8 9 9
0 0 3 4 5 8 8 12 12 12 12
```

- The maximum profit is 12.
- The items selected are 1, 2, and 4.
- The DP table is printed to show the step-by-step computation.