Fibonacci Code Explanation

def fibonacci_iter(n):
    if n < 0:
        return -1, 1
    if n == 0 or n == 1:
        return n, 1
    steps = 0
    a = 0
    b = 1
    for i in range(2, n+1):
        c = a + b
        a = b
        b = c
        steps += 1
    return c, steps+1

def fibonacci_recur(n):
    if n < 0:
        return -1, 1
    if n == 0 or n == 1:
        return n, 1
    fib1, steps1 = fibonacci_recur(n-1)
    fib2, steps2 = fibonacci_recur(n-2)
    return fib1 + fib2, steps1 + steps2 + 1

if __name__ == '__main__':
    n = int(input("Enter a number: "))
    print("Iterative:", fibonacci_iter(n)[0])
    print("Steps:", fibonacci_iter(n)[1])
    print("Recursive:", fibonacci_recur(n)[0])
    print("Steps:", fibonacci_recur(n)[1])

EXPLAIN CODE BY CODE LINE BY LINE

Sure! Let's go through the code line by line:

`fibonacci_iter(n)` function (Iterative Approach):

```python
def fibonacci_iter(n):
```

- This defines the function `fibonacci_iter`, which takes one parameter `n` (the index of the Fibonacci number you want to calculate).

```python
    if n < 0:
        return -1, 1
```

- This is a guard clause for invalid input. If `n` is less than 0, the function returns `-1` (indicating an error) and `1` (a placeholder for the step count).

```python
    if n == 0 or n == 1:
        return n, 1
```

- If `n` is either 0 or 1 (the base cases for Fibonacci numbers), the function returns the value of `n` (since Fibonacci of 0 is 0, and Fibonacci of 1 is 1) and `1` (indicating it took 1 step to calculate).

```python
    steps = 0
    a = 0
    b = 1
```

- Initializes `steps` (a counter for how many iterations the function makes), and sets `a` and `b` to represent the first two Fibonacci numbers (F(0) = 0 and F(1) = 1).

```python
    for i in range(2, n+1):
```

- A `for` loop that starts at 2 (since the first two Fibonacci numbers are already known) and goes until `n` (inclusive).

```python
        c = a + b
```

- In each iteration, the next Fibonacci number (`c`) is calculated by summing the previous two numbers (`a` and `b`).

```python
        a = b
```

```
        b = c
```

- Updates `a` to be the value of `b`, and `b` to be the newly calculated Fibonacci number `c`. This keeps track of the two most recent Fibonacci numbers.

```python
        steps += 1
```

- Increments the `steps` counter each time the loop runs, indicating one more iteration.

```python
    return c, steps+1
```

- After the loop ends, the function returns the final value of `c` (the Fibonacci number at index `n`) and the total number of steps, which is `steps + 1` (because the loop started from 2, so the count of steps is incremented after the last loop iteration).

`fibonacci_recur(n)` function (Recursive Approach):

```python
def fibonacci_recur(n):
```

- This defines the function `fibonacci_recur`, which calculates the Fibonacci number at index `n` using recursion.

```python
    if n < 0:
        return -1, 1
```

- This checks for invalid input, similar to the iterative function. If `n` is less than 0, it returns `-1` (error) and `1` (step count).

```python
    if n == 0 or n == 1:
        return n, 1
```

- This checks if `n` is either 0 or 1. If so, it directly returns `n` (since Fibonacci(0) = 0 and Fibonacci(1) = 1) and `1` (one recursive step).

```python
    fib1, steps1 = fibonacci_recur(n-1)
    fib2, steps2 = fibonacci_recur(n-2)
```

- These two lines recursively call `` `fibonacci_recur(n-1)` `` and `` `fibonacci_recur(n-2)` ``, which will keep reducing `` `n` `` until it reaches the base cases. `` `fib1` `` and `` `fib2` `` store the Fibonacci numbers from these recursive calls, and `` `steps1` `` and `` `steps2` `` store the number of recursive steps from those calls.

```python
    return fib1 + fib2, steps1 + steps2 + 1
```

- The result of `` `fibonacci_recur(n)` `` is the sum of the Fibonacci numbers from `` `n-1` `` and `` `n-2` ``. The total step count is the sum of `` `steps1` `` and `` `steps2` `` (from the recursive calls), plus `` `1` `` for the current step (the current function call itself).

Main Execution Block:

```python
if __name__ == '__main__':
```

- This ensures that the code inside this block only runs if the script is executed directly (not imported as a module).

```python
    n = int(input("Enter a number: "))
```

- Prompts the user to input a number `` `n` `` and converts the input to an integer.

```python
    print("Iterative:", fibonacci_iter(n)[0])
    print("Steps:", fibonacci_iter(n)[1])
```

- Calls the `` `fibonacci_iter(n)` `` function, which returns a tuple. The first element of the tuple is the Fibonacci number, and the second element is the number of steps. These values are printed separately.

```python
    print("Recursive:", fibonacci_recur(n)[0])
```

```
    print("Steps:", fibonacci_recur(n)[1])
```

- Similarly, calls the `fibonacci_recur(n)` function, which also returns a tuple. The first element is the Fibonacci number, and the second is the step count. These are printed as well.

Summary:

- The code implements two methods for calculating the Fibonacci number at index `n`: iteratively (`fibonacci_iter`) and recursively (`fibonacci_recur`).
- Both functions return the Fibonacci number and the number of steps taken to calculate it.
- The program asks the user for input and then displays the results for both methods.

The iterative approach is more efficient in terms of time complexity (O(n)), while the recursive approach is less efficient (O(2^n)) due to repeated calculations for the same Fibonacci values.

TELL ME TIME AND SPACE COMPLEXITY

Let's analyze the time and space complexity for both the iterative and recursive approaches for calculating the Fibonacci number.

1. Iterative Approach (`fibonacci_iter`)

Time Complexity:

- The iterative function uses a `for` loop that runs from 2 to `n`, meaning the loop runs `n-1` times.
- Inside the loop, there are constant-time operations (addition, assignment), so the time complexity is proportional to the number of iterations.

Thus, the **time complexity** is **O(n)**.

Space Complexity:

- The iterative function uses a constant amount of space. It only stores a few variables (`a`, `b`, `c`, and `steps`), all of which take up constant space regardless of the input size `n`.
- There are no data structures (like arrays or lists) that scale with `n`, so the space complexity is **O(1)**.

2. Recursive Approach (`fibonacci_recur`)

Time Complexity:

- The recursive function calls itself twice for each value of `n` (i.e., `fibonacci_recur(n-1)` and `fibonacci_recur(n-2)`).

- This results in an exponential growth of the number of calls. For each call, it further makes two recursive calls, leading to a total of approximately **2^n** calls.
- Since each call involves constant-time operations (adding two numbers, returning values), the **time complexity** is **O(2^n)**.

Space Complexity:

- The space complexity of the recursive approach is determined by the depth of the recursive call stack.
- The recursion goes down to the base case (`n == 0` or `n == 1`), and each recursive call adds a new frame to the call stack. In the worst case, the maximum depth of the call stack is proportional to `n`.
- Therefore, the **space complexity** due to the recursion stack is **O(n)**.

Summary:

- **Iterative Approach:**
  - Time Complexity: **O(n)**
  - Space Complexity: **O(1)**
- **Recursive Approach:**
  - Time Complexity: **O(2^n)**
  - Space Complexity: **O(n)**

The iterative approach is much more efficient in both time and space compared to the recursive approach.

WHAT IS ITERATIVE AND RECURSIVE APPROACH

Iterative Approach

In the **iterative approach**, we solve a problem using a loop (such as a `for` or `while` loop). The solution is built up step-by-step in a linear or repetitive manner. The loop iterates through a series of steps, gradually converging towards the final result.

Characteristics of the Iterative Approach:

- **Use of loops**: The problem is solved by repeating a set of operations until a certain condition is met.
- **State updates**: Variables are updated during each iteration of the loop to move closer to the result.
- **Efficiency**: Often more space- and time-efficient compared to recursion because it avoids the overhead of function calls and stack usage.

Example:

For calculating Fibonacci numbers, the iterative approach computes the Fibonacci numbers by iterating through the sequence from 2 to `n`, updating two variables at each step to hold the last two Fibonacci numbers.

```python
def fibonacci_iter(n):
    a, b = 0, 1
    for i in range(2, n+1):
        c = a + b
        a = b
        b = c
    return b
```

- **Time complexity**: O(n) because there is a single loop running from 2 to `n`.
- **Space complexity**: O(1) because it uses only a fixed number of variables.

Recursive Approach

In the **recursive approach**, the solution to a problem depends on solving smaller instances of the same problem. The function calls itself, and these calls continue until a base case (a simple, non-recursive case) is reached, at which point the results are "unwound" and combined to form the final solution.

Characteristics of the Recursive Approach:

- **Function calls**: The function calls itself with modified arguments, breaking down the problem into smaller subproblems.
- **Base case**: The recursion stops when the problem size becomes trivial (i.e., the base case).
- **Overhead**: Each recursive call adds a new layer to the function call stack, which can lead to more memory usage and slower performance if the problem size is large.

Example:

For calculating Fibonacci numbers, the recursive approach defines the Fibonacci function as a recursive call that breaks the problem down into two smaller Fibonacci calculations (`n-1` and `n-2`).

```python
def fibonacci_recur(n):
    if n == 0 or n == 1:
        return n
    return fibonacci_recur(n-1) + fibonacci_recur(n-2)
```

- **Time complexity**: O(2^n) because the function calls itself twice for each non-base case, resulting in exponential growth of the recursive calls.
- **Space complexity**: O(n) because each recursive call adds a new frame to the call stack, and the maximum recursion depth is `n`.

Comparison:

- **Iterative Approach**:
  - Solves the problem using loops.
  - Generally more efficient (especially in terms of space).
  - No overhead from function calls.
  - Best for problems where the number of iterations is predictable and finite.

- **Recursive Approach**:
  - Solves the problem by calling the function itself.
  - Can be elegant and easy to understand for problems that naturally break down into subproblems (like tree traversals or divide-and-conquer problems).
  - Has overhead due to function calls and the recursion stack.
  - Best suited for problems where the problem structure is inherently recursive (e.g., tree-like structures, problems that break down into similar subproblems).

Which one to use?

- **Use iterative** when the problem can be solved with loops without needing to break it into smaller subproblems.
- **Use recursive** when the problem naturally breaks down into smaller subproblems, especially if the recursion tree is manageable and doesn't involve repeated calculations (for large `n`, recursion can become inefficient without optimization like memoization).

```
import heapq

class node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right
        self.huff = ""

    def __lt__(self, other):
        return self.freq < other.freq

def printNodes(node, val=""):
    newval = val + node.huff
    if node.left:
        printNodes(node.left, newval)
    if node.right:
        printNodes(node.right, newval)
    else:
        print(f"{node.symbol} -> {newval}")

chars = ["a", "b", "c", "d", "e", "f"]
freqs = [5, 9, 12, 13, 16, 45]
```

nodes = []

for i in range(len(chars)):
    heapq.heappush(nodes, node(freqs[i], chars[i]))

while len(nodes) > 1:
    left = heapq.heappop(nodes)
    right = heapq.heappop(nodes)
    left.huff = "0"
    right.huff = "1"
    newnode = node(left.freq + right.freq, left.symbol + right.symbol, left, right)
    heapq.heappush(nodes, newnode)

printNodes(nodes[0])

EXPLAIN THIS CODE LINE BY LINE CODE BY CODE

Sure! Let me explain the code line by line.

1. Importing the heapq module

```python
import heapq
```

- The `heapq` module in Python provides an implementation of the **heap queue algorithm** (also known as the priority queue algorithm). It is used for efficiently managing a list of items where the smallest (or largest) item is always accessible in constant time.

2. Node class definition

```python
class node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right
        self.huff = ""
```

- A class called `node` is defined to represent a single node in the Huffman tree.
- The `__init__` method is the constructor for the node class. It initializes:
  - `freq`: Frequency of the symbol (used in the Huffman coding process).
  - `symbol`: The actual symbol (e.g., a character like 'a').
  - `left` and `right`: These represent the left and right child nodes of the current node in the tree.

- `huff`: A string that will store the Huffman code assigned to the symbol (initialized as an empty string).

```python
    def __lt__(self, other):
        return self.freq < other.freq
```

- The `__lt__` method defines a comparison for the `node` class using the less-than (`<`) operator.
- This method ensures that nodes are compared based on their `freq` attribute. It is important for maintaining the heap property when nodes are pushed and popped from the heap.

3. Defining the `printNodes` function

```python
def printNodes(node, val=""):
    newval = val + node.huff
```

- `printNodes` is a recursive function to print the Huffman codes for each symbol.
- It takes two arguments: `node` (the current node of the Huffman tree) and `val` (the Huffman code so far).
- `newval = val + node.huff`: It appends the current node's Huffman code (`node.huff`) to the `val` string, creating a new string `newval` that represents the path to the current node.

```python
    if node.left:
        printNodes(node.left, newval)
    if node.right:
        printNodes(node.right, newval)
    else:
        print(f"{node.symbol} -> {newval}")
```

- If the node has a left child, the function recursively calls `printNodes` with the left child and the updated `newval`.
- Similarly, if the node has a right child, it recursively calls `printNodes` with the right child and the updated `newval`.
- If the node is a leaf node (it doesn't have left or right children), it prints the symbol and the Huffman code (`newval`), which is the path from the root to that symbol.

4. Defining the characters and frequencies

```python
chars = ["a", "b", "c", "d", "e", "f"]
freqs = [5, 9, 12, 13, 16, 45]
```

- `chars` is a list of characters (symbols) that we want to assign Huffman codes to.

- `freqs` is a list of their corresponding frequencies.

5. Creating the initial heap (priority queue)

```python
nodes = []
for i in range(len(chars)):
    heapq.heappush(nodes, node(freqs[i], chars[i]))
```

- `nodes` is an empty list that will hold the nodes of the Huffman tree.
- The `for` loop iterates over the characters and frequencies. For each character-frequency pair, a `node` object is created with those values and pushed onto the `nodes` list using `heapq.heappush`.
  - The `heapq.heappush` function inserts a node into the heap, maintaining the heap property (the node with the lowest frequency will be at the root of the heap).

6. Building the Huffman Tree

```python
while len(nodes) > 1:
    left = heapq.heappop(nodes)
    right = heapq.heappop(nodes)
    left.huff = "0"
    right.huff = "1"
    newnode = node(left.freq + right.freq, left.symbol + right.symbol, left, right)
    heapq.heappush(nodes, newnode)
```

- The `while` loop continues until there is only one node left in the heap. This node will be the root of the Huffman tree.

- Inside the loop:
  1. **Extract two nodes**: The two nodes with the smallest frequencies are popped from the heap using `heapq.heappop`.
     - These will be the left and right children of the new internal node.
  2. **Assign Huffman codes**:
     - The left node is assigned the Huffman code `"0"`, and the right node is assigned the Huffman code `"1"`.
  3. **Create a new internal node**: A new `node` is created with the sum of the frequencies of the two extracted nodes (`left.freq + right.freq`), a combined symbol (`left.symbol + right.symbol`), and the left and right child nodes.
  4. **Push the new node back into the heap**: The new node is pushed back into the heap using `heapq.heappush`.

- This process repeats until the heap contains only one node, which will be the root of the Huffman tree.

7. Printing the Huffman Codes

```python
printNodes(nodes[0])
```

- After the Huffman tree is built, `nodes[0]` is the root of the tree.
- The `printNodes` function is called with the root node to print the Huffman codes for each symbol in the tree.

Explanation of the Huffman Tree Construction:

1. Start with a heap of nodes, each representing a character and its frequency.
2. Repeatedly remove the two nodes with the lowest frequencies and combine them into a new internal node.
3. Assign Huffman codes: "0" for left branches and "1" for right branches.
4. Push the new internal node back into the heap.
5. Continue this process until only one node remains in the heap (the root of the Huffman tree).
6. Finally, the `printNodes` function prints the Huffman code for each symbol.

Summary of Execution:

The code builds a **Huffman tree** from a set of characters and their frequencies and then prints the corresponding Huffman codes for each character. This algorithm is used for **lossless data compression**, where frequently occurring symbols are assigned shorter codes and less frequent ones are assigned longer codes.

TIME AND SPACE COMPLEXITY

Let's analyze the **time complexity** and **space complexity** of the Huffman coding algorithm, as implemented in the code.

1. Time Complexity

The main operations in the code are related to building the Huffman tree using a **min-heap** (priority queue) and creating the final Huffman codes.

Step 1: Building the Min-Heap (Initial Heap Construction)

```python
for i in range(len(chars)):
    heapq.heappush(nodes, node(freqs[i], chars[i]))
```

- You are inserting `n` nodes (where `n` is the number of characters) into the heap.

- The `heapq.heappush` operation runs in **O(log n)** time because each insertion into the heap requires reordering the heap, which takes logarithmic time.
- Therefore, the total time complexity for this step is **O(n log n)**.

Step 2: Building the Huffman Tree (Heap Operations)

```python
while len(nodes) > 1:
    left = heapq.heappop(nodes)
    right = heapq.heappop(nodes)
    heapq.heappush(nodes, newnode)
```

- In this loop, you are combining the two nodes with the smallest frequencies and inserting a new internal node back into the heap.

- Each iteration of the `while` loop performs the following operations:
  - **Popping two nodes**: `heapq.heappop` runs in **O(log n)** time. Since you are doing this twice in each iteration, the total time for both pops is **O(log n)**.
  - **Pushing the new node back into the heap**: `heapq.heappush` also takes **O(log n)** time.

- There are **n - 1** iterations in the `while` loop (because you keep merging nodes until only one node remains in the heap).

- Thus, the total time complexity for the while loop is **O(n log n)**.

Step 3: Printing the Huffman Codes

```python
printNodes(nodes[0])
```

- The `printNodes` function traverses the entire Huffman tree to print the codes. This is a depth-first traversal (DFS).
- The total number of nodes in the tree is `n` (since each symbol is represented by a node in the tree).
- Therefore, the time complexity for printing the Huffman codes is **O(n)**.

Overall Time Complexity

The overall time complexity is dominated by the steps that build the heap and the tree, both of which are **O(n log n)**. Thus, the total time complexity of the algorithm is:

$$O(n \log n)$$

2. Space Complexity

Step 1: Space for Nodes

```python
nodes = []
```

- You are storing a list of `n` nodes (each node represents a character and its frequency), so the space complexity for the nodes is **O(n)**.

Step 2: Space for Huffman Tree

- The Huffman tree itself also contains `n` nodes, and since each node stores references to its left and right children, this contributes an additional **O(n)** space.

Step 3: Space for Recursive Call Stack (for printNodes)

- The `printNodes` function uses recursion. In the worst case, it might recurse all the way to the leaf nodes, resulting in a recursion depth equal to the height of the tree.
- Since the height of a balanced Huffman tree is **O(log n)**, the space complexity due to the recursion stack is **O(log n)**.

Overall Space Complexity

The space complexity is dominated by the storage of the `n` nodes in the heap and the Huffman tree. Thus, the total space complexity of the algorithm is:

$$O(n)$$

Summary

- **Time Complexity**: **O(n log n)**, due to the heap operations (insertions and deletions) and the tree-building process.
- **Space Complexity**: **O(n)**, for storing the nodes of the Huffman tree and the heap.

This makes Huffman coding efficient in both time and space, particularly when the number of symbols (characters) is large.

```
class Item:
    def __init__(self, profit, weight):
        self.profit = profit
        self.weight = weight

def fractionalKnapsack(w, arr):
    arr.sort(key=lambda x: x.profit/x.weight, reverse=True)
    finalValue = 0.0
    for item in arr:
        if w >= item.weight:
            finalValue += item.profit
            w -= item.weight
```

```
        else:
            finalValue += item.profit * (w/item.weight)
            break
    return finalValue

if __name__ == "__main__":
    n = int(input("Enter number of items-\n"))
    arr = []
    for i in range(n):
        profit = int(input("Enter profit of item " + str(i + 1) + "-\n"))
        weight = int(input("Enter weight of item " + str(i + 1) + "-\n"))
        arr.append(Item(profit, weight))
    w = int(input("Enter capacity of knapsack-\n"))
    print("Maximum value in knapsack: ", fractionalKnapsack(w, arr))
```

EXPLAIN THIS CODE LINE BY LINE - CODE BY CODE

Let's go through the code step by step to understand it in detail.

1. Class Definition: `Item`

```python
class Item:
    def __init__(self, profit, weight):
        self.profit = profit
        self.weight = weight
```

- A class `Item` is defined to represent an item that can be placed in a knapsack.
- The constructor (`__init__`) takes two arguments: `profit` (the profit of the item) and `weight` (the weight of the item).
- The `profit` and `weight` values are stored as attributes of the object (`self.profit` and `self.weight`).

2. Fractional Knapsack Function: `fractionalKnapsack`

```python
def fractionalKnapsack(w, arr):
```

- This function calculates the maximum profit that can be achieved using the fractional knapsack problem.
- `w` is the **capacity** of the knapsack.
- `arr` is the list of `Item` objects, representing the items available for selection.

2.1 Sorting Items by Profit-to-Weight Ratio

```python
arr.sort(key=lambda x: x.profit/x.weight, reverse=True)
```

- Here, the list `arr` (which contains `Item` objects) is sorted in descending order by the **profit-to-weight ratio**.
- The lambda function `lambda x: x.profit/x.weight` is used to calculate the ratio for each item.
  - `x.profit` refers to the profit of the item.
  - `x.weight` refers to the weight of the item.
- Sorting in **reverse=True** ensures that the item with the highest profit-to-weight ratio comes first, as this is the greedy approach used to maximize profit in fractional knapsack.

2.2 Initializing Final Value

```python
finalValue = 0.0
```

- `finalValue` will hold the total profit accumulated by filling the knapsack. It is initialized to 0.0, as no items have been added to the knapsack yet.

2.3 Iterating Through the Sorted List

```python
for item in arr:
    if w >= item.weight:
        finalValue += item.profit
        w -= item.weight
    else:
        finalValue += item.profit * (w/item.weight)
        break
```

- The loop iterates through the sorted list of items.
- Inside the loop:
  1. **If the current item's weight fits in the remaining capacity of the knapsack** (`w >= item.weight`):
     - The entire item can be taken. Add its full profit to `finalValue`, and subtract its weight from the knapsack's remaining capacity (`w -= item.weight`).
  2. **If the current item's weight exceeds the remaining capacity of the knapsack** (`else`):
     - Take a fractional part of the item to fit the knapsack. The fraction of the item taken is proportional to the remaining capacity, so the profit added is calculated as `item.profit * (w/item.weight)`.
     - The loop then breaks because the knapsack is full, and no more items can be added.

2.4 Returning the Final Value

```python
    return finalValue
```

- Once the loop completes, the total accumulated profit (`finalValue`) is returned as the result of the function.

3. Main Function Execution

```python
if __name__ == "__main__":
```

- This checks if the script is being run directly (not imported as a module).
- If this condition is `True`, the code inside this block will be executed.

3.1 Input for Number of Items

```python
    n = int(input("Enter number of items-\n"))
```

- `n` is the number of items. The program prompts the user to enter the number of items (`n`) they want to add to the knapsack.

3.2 Creating List of Items

```python
    arr = []
    for i in range(n):
        profit = int(input("Enter profit of item " + str(i + 1) + "-\n"))
        weight = int(input("Enter weight of item " + str(i + 1) + "-\n"))
        arr.append(Item(profit, weight))
```

- An empty list `arr` is initialized to store the `Item` objects.
- The `for` loop runs `n` times, asking the user to input the **profit** and **weight** for each item.
- For each item, a new `Item` object is created using the input values (`profit` and `weight`), and it is appended to the list `arr`.

3.3 Input for Knapsack Capacity

```python
    w = int(input("Enter capacity of knapsack-\n"))
```

- The user is prompted to input the **capacity** of the knapsack (`w`), which determines how much weight the knapsack can hold.

3.4 Calling `fractionalKnapsack` Function and Displaying the Result

```python
print("Maximum value in knapsack: ", fractionalKnapsack(w, arr))
```

- The `fractionalKnapsack` function is called with the knapsack capacity `w` and the list of `Item` objects `arr`.
- The function returns the **maximum value** (profit) that can be carried in the knapsack.
- The result is printed to the console.

Explanation of the Algorithm:

- This code implements the **Fractional Knapsack problem**, which is a greedy algorithm for selecting items (or fractions of items) to maximize profit while staying within a given weight capacity.
- The items are first sorted by their **profit-to-weight ratio** in descending order. The algorithm then adds items to the knapsack, starting with the item with the highest ratio. If an item doesn't fit fully, a fractional part of it is added to fill the remaining capacity.

Example:

If the user enters:

- Number of items: 4
- Item 1: Profit = 60, Weight = 10
- Item 2: Profit = 100, Weight = 20
- Item 3: Profit = 120, Weight = 30
- Knapsack capacity: 50

The items would be sorted by the profit-to-weight ratio, and the algorithm will try to fit them into the knapsack starting with the item with the highest ratio.

Time Complexity:

- Sorting the items takes **O(n log n)** because of the sort operation.
- Iterating through the items to fill the knapsack takes **O(n)**.
- Overall time complexity: **O(n log n)** (dominated by the sorting step).

Space Complexity:

- The space complexity is **O(n)** because we store the items in a list of size `n`.

Summary:

This is a greedy algorithm to solve the fractional knapsack problem, which aims to maximize profit by choosing items based on their profit-to-weight ratio, allowing for fractional items to be taken when necessary.

```
def knapsack_01(n, values, weights, W):
    dp = [[0] * (W+1) for _ in range(n+1)]

    for i in range(n+1):
        for w in range(W+1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]

    selected_items = []
    i, w = n, W
    while i > 0 and w > 0:
        if dp[i][w] != dp[i-1][w]:
            selected_items.append(i-1)
            w -= weights[i-1]
        i -= 1

    return dp[n][W], selected_items

if __name__ == "__main__":
    n = 3
    values = [60, 100, 120]
    weights = [10, 20, 30]
    W = 50

    max_value, selected_items = knapsack_01(n, values, weights, W)
    print("Maximum value:", max_value)
    print("Selected items:", selected_items)
```

EXPLAIN CODE BY CODE LINE BY LINE

Let's break down the code step by step to understand how it works.

1. `knapsack_01` Function Definition

```python
def knapsack_01(n, values, weights, W):
```

- The `knapsack_01` function implements the **0/1 Knapsack problem** using dynamic programming (DP).
- **Parameters:**
  - `n`: The number of items.
  - `values`: A list of item values (profits).
  - `weights`: A list of item weights.
  - `W`: The capacity of the knapsack.

2. DP Table Initialization

```python
    dp = [[0] * (W+1) for _ in range(n+1)]
```

- A 2D array `dp` is created, with dimensions `(n+1) x (W+1)`. This array will store the maximum profit achievable with the first `i` items and a knapsack capacity of `w`.
  - `dp[i][w]` will store the maximum profit for the first `i` items with a knapsack capacity of `w`.
  - The array is initialized to 0, meaning no items selected and no profit.

3. Populating the DP Table

```python
    for i in range(n+1):
        for w in range(W+1):
            if i == 0 or w == 0:
                dp[i][w] = 0
```

- This double `for` loop goes through each combination of items (from 0 to `n`) and capacities (from 0 to `W`).
- **Base Case**: If there are no items (`i == 0`) or the knapsack has zero capacity (`w == 0`), then the maximum profit is 0, as there are no items to select or no capacity to carry items.

```python
            elif weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
```

- **If the current item's weight is less than or equal to the current capacity (`weights[i-1] <= w`):**
  - The current item can be included in the knapsack, and we compute the maximum profit using two options:
    1. **Exclude the item**: The maximum value is `dp[i-1][w]` (profit from the previous row with the same capacity).
    2. **Include the item**: The maximum value is `dp[i-1][w-weights[i-1]] + values[i-1]` (profit from the previous row with the remaining capacity after including the current item).

- We take the **maximum** of these two options.

```python
            else:
                dp[i][w] = dp[i-1][w]
```

- **If the current item's weight exceeds the current capacity (`weights[i-1] > w`):**
  - The current item cannot be included, so the maximum value remains the same as excluding the item: `dp[i][w] = dp[i-1][w]`.

4. Tracing Back to Find the Selected Items

```python
    selected_items = []
    i, w = n, W
```

- After filling the DP table, we need to trace back the selected items (those included in the knapsack).
- We initialize `i = n` (the last item) and `w = W` (the full capacity of the knapsack).
- An empty list `selected_items` is created to store the indices of the items that are selected.

```python
    while i > 0 and w > 0:
        if dp[i][w] != dp[i-1][w]:
            selected_items.append(i-1)
            w -= weights[i-1]
```

- This `while` loop traces back through the DP table to find the items included in the knapsack.
- **If `dp[i][w]` is different from `dp[i-1][w]`:**
  - This means the current item `i-1` was included in the optimal solution, as the profit changes when including the item.
  - The index `i-1` is added to `selected_items` (since the list is 0-indexed).
  - The remaining capacity `w` is reduced by the weight of the current item (`w -= weights[i-1]`).

```python
        i -= 1
```

- Move to the previous item (`i -= 1`) and continue the process until `i` becomes 0 or `w` becomes 0, meaning no more items are left or the knapsack is full.

5. Return the Maximum Value and Selected Items

```python
    return dp[n][W], selected_items
```

- The function returns two values:
  - `dp[n][W]`: The maximum value (profit) that can be achieved with `n` items and a knapsack capacity `W`.
  - `selected_items`: A list of indices of the items selected in the optimal solution.

6. Main Function Execution

```python
if __name__ == "__main__":
```

- This block checks if the script is being run directly (not imported as a module). If `True`, the code inside the block will be executed.

6.1 Input Values for the Knapsack Problem

```python
    n = 3
    values = [60, 100, 120]
    weights = [10, 20, 30]
    W = 50
```

- In the main function:
  - `n = 3`: There are 3 items.
  - `values = [60, 100, 120]`: The values (profits) of the items.
  - `weights = [10, 20, 30]`: The weights of the items.
  - `W = 50`: The capacity of the knapsack.

6.2 Calling the `knapsack_01` Function

```python
    max_value, selected_items = knapsack_01(n, values, weights, W)
```

- The `knapsack_01` function is called with the number of items `n`, the `values`, the `weights`, and the knapsack capacity `W`.
- It returns the maximum value `max_value` and the list of `selected_items` (indices of the items selected in the optimal solution).

6.3 Printing the Results

```python
python


    print("Maximum value:", max_value)
    print("Selected items:", selected_items)
```

- The maximum profit that can be obtained with the given capacity is printed.
- The indices of the selected items are printed, showing which items are part of the optimal solution.

Example Run:

If the input is:

- `values = [60, 100, 120]`
- `weights = [10, 20, 30]`
- `W = 50`

The output will be:

- `Maximum value: 220`
- `Selected items: [2, 1]`

This means the maximum profit that can be obtained is 220 by selecting the items with indices `2` and `1` (items with values 120 and 100, and weights 30 and 20).

Time Complexity:

- **Filling the DP table**: There are `n` rows and `W` columns in the DP table. Each cell is filled in constant time, so the time complexity is **O(n * W)**.
- **Traceback to find the selected items**: In the worst case, we trace back `n` items, so this step has time complexity **O(n)**.

Therefore, the overall time complexity is **O(n * W)**.

Space Complexity:

- The DP table has dimensions `(n+1) x (W+1)`, so the space complexity is **O(n * W)**.
- The `selected_items` list stores at most `n` items, so the space complexity for this is **O(n)**.

Thus, the overall space complexity is **O(n * W)**.