Explanation of DAA codes:-

**A1**

the recursive approach involves problems solving themselves by breaking down into smaller versions of the same problem, while the non-recursive approach solves problems one step at a time without self-replicating steps.

1. The function takes an integer **n** as input, which represents the position of the Fibonacci number to be calculated.
2. It starts with a base case check. If **n** is less than or equal to 1, it returns **n** because the Fibonacci sequence starts with 0 and 1, and any number in the sequence at or before these positions is the same as its position.
3. If **n** is greater than 1, the function initializes two variables **a** and **b** to 0 and 1, respectively. These variables are used to keep track of the two most recent Fibonacci numbers.
4. It then enters a loop that iterates from 2 to **n**. Inside the loop, it updates **a** and **b** as follows:
   * **a** is set to the current value of **b**.
   * **b** is updated to the sum of the previous values of **a** and **b**, which effectively calculates the next Fibonacci number in the sequence.
5. After the loop is completed, the function returns the value of **b**, which is the nth Fibonacci number.

In your example usage, you calculate and print the 10th Fibonacci number, and the result is displayed as "The 10th Fibonacci number is 55." This is because the 10th Fibonacci number is indeed 55.

1. **import time**: This line imports the **time** module, which provides various time-related functions, including measuring the execution time of code.
2. **start = time.time()**: This line records the current time in seconds since the epoch (the time at which the system started running) and stores it in the variable **start**. This is done to measure the start time of the execution.
3. The **fibonacci\_iterative** function is the same as in your previous code, which calculates the nth Fibonacci number iteratively.
4. **n = 1000**: This line sets the value of **n** to 1000, specifying that you want to calculate the 1000th Fibonacci number.
5. **result = fibonacci\_iterative(n)**: This line calls the **fibonacci\_iterative** function with **n** as an argument to calculate the 1000th Fibonacci number and stores the result in the variable **result**.
6. **print(f"The {n}th Fibonacci number is {result}.")**: This line prints the result of the Fibonacci calculation, showing the position (**n**) and the calculated Fibonacci number (**result**).
7. **end = time.time()**: This line records the current time again after the Fibonacci calculation is complete and stores it in the variable **end**. This allows you to measure the end time of the execution.
8. **print("Execution time is: {}ms".format((end-start)\*10\*\*3))**: This line calculates the execution time by subtracting the **start** time from the **end** time and then multiplies the result by 1000 to convert it from seconds to milliseconds. It then prints the execution time in milliseconds.

The purpose of this code is to measure how long it takes to calculate the 1000th Fibonacci number using the **fibonacci\_iterative** function and print the result along with the execution time in milliseconds.

RECURSIVE APPROACH-

1. **def fibonacci\_recursive(n):** - This line defines a Python function called **fibonacci\_recursive** that takes an integer **n** as its parameter. This function is intended to calculate the nth Fibonacci number.
2. **if n <= 1:** - This is a base case check. If **n** is less than or equal to 1, the function returns **n**. This is because the Fibonacci sequence starts with 0 and 1, and any number in the sequence at or before these positions is the same as its position.
3. **return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)** - This is the recursive part of the function. If **n** is greater than 1, it calls the **fibonacci\_recursive** function twice with **n - 1** and **n - 2** as arguments and then returns the sum of these two function calls. This is how the function recursively calculates the nth Fibonacci number by breaking it down into smaller Fibonacci numbers until it reaches the base case.
4. The code then proceeds to an example usage:
   * **n = 10** - Sets the value of **n** to 10, indicating that you want to calculate the 10th Fibonacci number.
   * **result = fibonacci\_recursive(n)** - Calls the **fibonacci\_recursive** function with **n** as an argument to calculate the 10th Fibonacci number and stores the result in the variable **result**.
   * **print(f"The {n}th Fibonacci number is {result}.")** - Prints the result, indicating that it's the 10th Fibonacci number.

The **fibonacci\_recursive** function uses a recursive approach to calculate Fibonacci numbers. It breaks the problem down into smaller subproblems and relies on repeated function calls to calculate the final result. However, it can be less efficient for large values of **n** compared to non-recursive approaches due to repeated calculations.

1. **import time**: This line imports the **time** module, which provides various time-related functions, including measuring the execution time of code.
2. **start = time.time()**: This line records the current time in seconds since the epoch (the time at which the system started running) and stores it in the variable **start**. This is done to measure the start time of the execution.
3. The **fibonacci\_recursive** function is the same as in your previous code, which calculates the nth Fibonacci number using a recursive approach.
4. **n = 10**: This line sets the value of **n** to 10, indicating that you want to calculate the 10th Fibonacci number.
5. **result = fibonacci\_recursive(n)**: This line calls the **fibonacci\_recursive** function with **n** as an argument to calculate the 10th Fibonacci number and stores the result in the variable **result**.
6. **print(f"The {n}th Fibonacci number is {result}.")**: This line prints the result of the Fibonacci calculation, showing the position (**n**) and the calculated Fibonacci number (**result**).
7. **end = time.time()**: This line records the current time again after the Fibonacci calculation is complete and stores it in the variable **end**. This allows you to measure the end time of the execution.
8. **print("Execution time is: {}ms".format((end-start)\*10\*\*3))**: This line calculates the execution time by subtracting the **start** time from the **end** time and then multiplies the result by 1000 to convert it from seconds to milliseconds. It then prints the execution time in milliseconds.

The time complexity of this approach is exponential, O(2^n). This is because it recursively calls itself twice for each value of n, The space complexity is O(n) because it creates a recursive call stack of depth n.

**A3-**

1. **import time**: This line imports the **time** module, which provides various time-related functions, including measuring the execution time of code.
2. **start = time.time()**: This line records the current time in seconds since the epoch (the time at which the system started running) and stores it in the variable **start**. This is done to measure the start time of the overall program execution.
3. **def fractional\_knapsack(items, capacity):** - This line defines a Python function called **fractional\_knapsack** that takes two arguments: **items**, which is a list of items represented as tuples of weight and profit, and **capacity**, which represents the maximum weight the knapsack can hold.
4. **algo\_start = time.time()**: Inside the **fractional\_knapsack** function, this line records the current time to measure the start time of the algorithm execution.
5. **item\_value\_ratio = [(item[1] / item[0], item) for item in items]**: This line calculates the value-to-weight ratio for each item in the **items** list and stores both the ratio and the original item as a tuple in a new list called **item\_value\_ratio**. This is a key step for the fractional knapsack algorithm, as it helps prioritize items based on their value per unit weight.
6. **item\_value\_ratio.sort(reverse=True)**: It sorts the **item\_value\_ratio** list in descending order based on the calculated value-to-weight ratios. This step ensures that items with the highest value-to-weight ratio come first.
7. **total\_value = 0**: Initializes the variable **total\_value** to keep track of the total value of the selected items in the knapsack.
8. **knapsack = []**: Initializes the **knapsack** list to keep track of the items selected for the knapsack.
9. The following block of code iterates through the sorted **item\_value\_ratio** list and selects items to put in the knapsack based on their value-to-weight ratio and the available capacity:
   * If the entire item can fit in the knapsack (**capacity >= item[0]**), it adds the entire item to the **knapsack**, updates **total\_value**, and reduces **capacity**.
   * If the item can only fit partially, it calculates the fraction that can fit, adds a tuple of the item and the fraction to the **knapsack**, updates **total\_value** accordingly, and breaks the loop since the knapsack is full.
10. **algo\_end = time.time()**: After the knapsack algorithm execution is complete, it records the current time to measure the end time of the algorithm execution.
11. **algo\_exec = (algo\_end-algo\_start)\*10\*\*3**: It calculates the execution time of the **fractional\_knapsack** algorithm by subtracting the start time from the end time and then multiplying the result by 1000 to convert it from seconds to milliseconds.
12. The **fractional\_knapsack** function returns the **total\_value**, a list of selected items and their fractions in the knapsack, and the execution time of the algorithm in milliseconds.
13. The code then proceeds to an example usage, where the user is prompted to input the details of items, their weights, profits, and the knapsack's capacity.
14. It calculates the maximum value (**max\_value**), the selected items in the knapsack (**selected\_items**), and the algorithm's execution time (**algo\_execution\_time**) by calling the **fractional\_knapsack** function with the provided inputs.
15. It prints the maximum value and the details of the selected items, including the fraction of each item in the knapsack.
16. **end = time.time()**: Records the current time again after the program execution is complete to measure the end time of the overall program execution.
17. Finally, it calculates and prints the algorithm execution time and the program execution time (from the start of the program to its end) in milliseconds.

**A5-**

1. **import time**: We're importing the **time** module to work with time and measure how long the program takes to run.
2. **start = time.time()**: We're recording the current time to know when the program started running.
3. **def print\_solution(board):**: This is the start of a function definition named **print\_solution**. It's a function that takes a "board" as input and is responsible for printing the solution.
4. **for row in board:**: This line starts a loop that goes through each row in the "board."
5. **print(" ".join(row))**: For each row, it prints the content of that row, with each element separated by a space.
6. **print("\n")**: This prints a newline character to separate the rows visually.
7. **def is\_safe(board, row, col, n):**: This is the start of another function definition, **is\_safe**. It checks whether it's safe to place a queen in a particular position on the board. It takes the board, the row and column to be checked, and the size of the board as input.
8. The first **for** loop checks if there is any queen in the same column in the rows above the current row. If it finds a queen in the same column, it returns **False** because that would mean the placement is not safe.
9. The next two loops check if there are queens in the diagonals to the left and right of the current position. If a queen is found in any of these diagonals, the function returns **False** because that placement is not safe.
10. If no issues are found during the checks, the function returns **True**, indicating that it's safe to place a queen in the specified position.
11. **def solve\_n\_queens(board, row, n):**: This is another function definition named **solve\_n\_queens**. It is responsible for solving the N-Queens problem using backtracking. It takes the board, the current row, and the size of the board as input.
12. If the **row** is equal to **n**, it means that all queens have been placed successfully, and a solution has been found. In this case, it calls the **print\_solution** function to print the board with the queens placed and then returns.
13. The following loop goes through each column in the current row, checking if it's safe to place a queen there using the **is\_safe** function. If it's safe, the code places a queen on the board, calls itself recursively to continue to the next row, and then backtracks by removing the queen (represented by 'Q') if needed.
14. **def n\_queens(n):**: This is another function definition that is responsible for setting up the initial board and starting the solving process. It takes the number of queens (n) as input.
15. If the input **n** is less than or equal to 0, it returns an empty list because there is no valid solution.
16. It creates an **n** by **n** board filled with empty squares represented as **'.'** and calls the **solve\_n\_queens** function to solve the N-Queens problem.
17. The program starts by asking the user to input the number of queens they want to place on the board.
18. It then calls the **n\_queens** function with the user-provided value to solve the N-Queens problem.
19. After the problem is solved and the program has executed, it records the end time.
20. Finally, it calculates and prints the execution time for the entire program in milliseconds.

**A4-**

1. **import time**: We're including a tool to measure time during our program.
2. **start = time.time()**: We're noting the current time to remember when our program begins running.
3. **def knapsack\_dynamic\_programming(values, weights, capacity):**: This is the beginning of a function we're creating. It's like a set of instructions we can use. This function helps us solve the knapsack problem using dynamic programming. It takes three things as input: **values** (how valuable the items are), **weights** (how heavy the items are), and **capacity** (the maximum weight the knapsack can hold).
4. **algo\_start = time.time()**: We're recording the current time to measure when our knapsack-solving algorithm starts working.
5. **n = len(values)**: We're figuring out how many items we have by counting how many values are in our list.
6. We're creating a table called **dp** to store values for solving the knapsack problem using dynamic programming. This table has **n + 1** rows (for each item) and **capacity + 1** columns (for each possible weight the knapsack can have).
7. This part of the code uses two loops:
   * The outer loop goes through each item, from the first to the last, plus an extra row for when there are no items (i.e., 0 items).
   * The inner loop goes through each possible weight the knapsack can have, from 0 to the maximum weight it can hold.
8. If we're in the first row (no items) or the first column (no capacity), we set the value in the table to 0 because, with no items or no capacity, the maximum value you can get is 0.
9. If the weight of the current item is less than or equal to the current capacity we're considering, we have a choice. We can either take the current item and add its value to the maximum value we can get with the remaining capacity, or we can choose not to take the item and stick with the maximum value we can get without it.
10. If the weight of the current item is greater than the current capacity, we have no choice but to skip the item. In this case, we set the value in the table to the maximum value we can get without this item, which is the value above it in the table.
11. We're now keeping track of which items to put in the knapsack, and we're remembering the current row and column in our table.
12. We're going backward in our table to find the items we've put in our knapsack by checking which items affect the value we've calculated.
13. After finding the items, we reverse the order because they were added in reverse order during backtracking.
14. We're recording the current time to know when our algorithm ends running.
15. We're figuring out how long our algorithm took to run, and we're measuring the time in milliseconds.
16. Our **knapsack\_dynamic\_programming** function returns the maximum value you can get, the items you've put in your knapsack, and the time it took for the algorithm to run.
17. We're continuing with an example: you tell the program how many items you have, and then you input the values and weights for those items, as well as the capacity of your knapsack.
18. The program calculates and shows the maximum value you can get and which items you should put in your knapsack.
19. We're recording the current time to know when our program ends running.
20. We're figuring out how long our program took to run, and we're measuring the time in milliseconds. This helps us see how fast our program is.

**A2-**



above answer in short

ChatGPT

Huffman coding is a method for data compression. It assigns shorter codes to frequently occurring symbols, reducing the size of data. It creates a binary tree where the path to each symbol determines its code. This efficient encoding and decoding method is widely used for file compression and data transmission.



greedy approach in brief

ChatGPT

A greedy approach is a problem-solving strategy that involves making the locally optimal choice at each step with the hope of finding a global optimum (the best overall solution). In a greedy algorithm:

1. You start with an empty solution or an initial solution.
2. At each step, you select the best available option based on a certain criterion.
3. You continue making these choices one step at a time until you reach a solution.

The key idea is that at each step, a greedy algorithm makes the choice that seems best at that moment without considering the consequences of that choice on future steps. Greedy algorithms are often used when solving optimization problems where you want to maximize or minimize a certain value (e.g., maximizing profit or minimizing cost) while adhering to specific constraints.

Greedy algorithms are known for their simplicity and efficiency but may not always guarantee an optimal solution for every problem. Whether a greedy approach works depends on the problem's nature and whether the greedy choice rule leads to the best global solution. In some cases, greedy algorithms produce the best result, while in others, a more complex approach is required to ensure an optimal outcome.

This code is an implementation of Huffman coding, a popular algorithm for lossless data compression. It creates a Huffman tree and generates Huffman codes for a set of characters based on their frequencies. Let's break down the code line by line in simple terms:

1. **class Node:** defines a class for nodes in the Huffman tree. Each node has properties for frequency, symbol, left child, right child, and a "huff" value to represent the direction (0 or 1) in the tree.
2. **def \_\_init\_\_(self, freq\_, symbol\_, left\_=None, right\_=None):** is the constructor method for the Node class. It initializes the node with its frequency, symbol (character), and optional left and right children.
3. **def print\_nodes(node, val=""):** defines a utility function to print the Huffman codes. It takes a node and an optional value as input, which represents the current code being constructed.
4. **new\_val = val + str(node.huff)** constructs the new code by appending the current node's "huff" value to the existing code.
5. The following conditional statements:
   * **if node.left:** and **if node.right:** check if the node has left and right children. If so, it recursively calls **print\_nodes** on them with the updated **new\_val**.
6. **if not node.left and not node.right:** checks if the current node is a leaf node (it has no children). If so, it prints the character and its corresponding Huffman code.
7. The list **chars** contains the characters to be encoded, and **freq** contains their corresponding frequencies.
8. **nodes** is a list of Node objects created from the characters and their frequencies. Each Node represents a character with its frequency.
9. The following while loop continues until there is only one node left in the **nodes** list:
   * It sorts the **nodes** list based on the frequencies of the nodes in ascending order using the **sorted** function and a lambda function as the sorting key.
   * It selects the two smallest nodes, **left** and **right**, from the sorted list.
   * It sets the **huff** value of the **left** node to 0 and the **right** node to 1 to represent their position in the Huffman tree.
   * It creates a new node, **newNode**, as the parent of the **left** and **right** nodes, with a symbol that combines the symbols of its children.
   * It removes the **left** and **right** nodes from the **nodes** list and adds the **newNode** to it.
10. Finally, the code prints the characters and their frequencies, followed by the generated Huffman codes by calling the **print\_nodes** function on the root node of the Huffman tree (the last node in the **nodes** list).

In summary, this code constructs a Huffman tree for a set of characters and their frequencies, and it prints the Huffman codes for each character. These Huffman codes are used for data compression, where frequently occurring characters have shorter codes, reducing the overall size of the data.

**A6-**



explain quicksort algorithm with example

ChatGPT

Sure, I'd be happy to explain the Quick Sort algorithm with an example.

Quick Sort is a widely used sorting algorithm that follows the "divide and conquer" approach. It works as follows:

1. **Choose a Pivot Element**: Pick a "pivot" element from the array. The choice of the pivot can affect the performance of the algorithm, but for simplicity, let's say we choose the last element in the array as the pivot.
2. **Partitioning**: Rearrange the array so that all elements less than the pivot are to its left, and all elements greater than the pivot are to its right. The pivot itself will be in its final sorted position.

Let's take an example array for better understanding:

pythonCopy code

Original array: [7, 2, 1, 6, 8, 5, 3, 4] Pivot element: 4 (last element)

After partitioning, the array might look like this:

pythonCopy code

[2, 1, 3, 4, 8, 5, 7, 6]

The pivot (4) is in its final sorted position, and elements less than it are on the left, and elements greater than it are on the right.

1. **Recursion**: Apply the Quick Sort algorithm recursively to the subarrays on the left and right of the pivot. This means we sort the left subarray (**[2, 1, 3]**) and the right subarray (**[8, 5, 7, 6]**) separately.
2. **Combining**: Once the subarrays are sorted, you combine them with the pivot in the middle to obtain the final sorted array.

In this case, the sorted array would be:

pythonCopy code

[1, 2, 3, 4, 5, 6, 7, 8]

Certainly! Let's explain Deterministic Quick Sort with an example.

Suppose you have the following unsorted array of numbers:

pythonCopy code

[5, 1, 9, 3, 7, 6]

In Deterministic Quick Sort:

1. The algorithm always selects the first element of the array as the pivot. So, in this case, the pivot is **5**.
2. The algorithm then rearranges the elements in the array so that all elements less than or equal to the pivot (5) are on the left, and all elements greater than the pivot are on the right.

After this step, your array might look like this:

csharpCopy code

[1, 3, 5, 9, 7, 6]

1. The pivot element, **5**, is now in its final sorted position.
2. The algorithm then applies the same process to the subarrays on the left and right of the pivot. It selects a new pivot for each subarray (e.g., **1** on the left and **9** on the right) and repeats the process.
3. The process continues recursively until the entire array is sorted.

In this case, the algorithm will sort the left subarray (**[1, 3]**) and the right subarray (**[9, 7, 6]**) separately.

1. Once all subarrays are sorted, the entire array is sorted.

The sorted array would be **[1, 3, 5, 6, 7, 9]**.

While Deterministic Quick Sort is generally efficient and widely used, it has some weaknesses, especially when the input data is already sorted or nearly sorted, as it can result in poor performance in those cases. This is why randomized variants of Quick Sort, which choose the pivot element randomly, are often used to improve the algorithm's performance on various types of input data.

This code performs a comparison between deterministic and randomized versions of the Quick Sort algorithm on a given array of numbers. Here's a breakdown of the code line by line in simple terms:

1. **import random** and **import time** import the required Python modules for generating random numbers and measuring time.
2. **def quick\_sort(arr):** defines a function for deterministic Quick Sort. It takes an unsorted array **arr** as input and returns a sorted version of the array.
3. Inside **quick\_sort**, if the length of the array is 1 or less, it returns the array as it is (base case for recursion).
4. The pivot element is chosen as the first element of the array (a deterministic choice).
5. The **left** list contains elements less than or equal to the pivot, and the **right** list contains elements greater than the pivot.
6. The function recursively calls **quick\_sort** on the **left** and **right** subarrays and combines them with the pivot element to return the sorted array.
7. **def randomized\_quick\_sort(arr):** defines a function for randomized Quick Sort, which is similar to **quick\_sort** but with a random pivot selection. It chooses a pivot randomly from the input array.
8. **def analyze\_quick\_sort(arr, variant="deterministic"):** is a function for analyzing and comparing the two Quick Sort variants. It takes the input array **arr** and an optional **variant** argument, which can be "deterministic" or "randomized."
9. Inside **analyze\_quick\_sort**, it records the start time.
10. Depending on the **variant** specified, it calls either **quick\_sort** or **randomized\_quick\_sort** to sort the array.
11. It calculates the execution time by subtracting the start time from the current time.
12. It returns the sorted array and the execution time.
13. The code collects input from the user to create an array. It asks for the number of elements and the elements themselves.
14. It prints the original unsorted array.
15. It calls **analyze\_quick\_sort** for both the deterministic and randomized variants and records the sorted arrays and execution times.
16. Finally, it prints the results, showing the sorted arrays and the execution times for both variants.

In summary, this code compares the performance of deterministic and randomized Quick Sort on a user-provided array, measuring the execution time for each and displaying the sorted arrays and execution times. The randomized variant selects a random pivot element, which can improve performance in some cases compared to always using the first element as the pivot (as in the deterministic variant).



left = [x for x in arr[:pivot\_index] + arr[pivot\_index+1:] if x <= pivot] right = [x for x in arr[:pivot\_index] + arr[pivot\_index+1:] if x > pivot] return randomized\_quick\_sort(left) + [pivot] + randomized\_quick\_sort(right) explain in simple words

ChatGPT

Certainly! These lines of code are part of the randomized version of the Quick Sort algorithm, and they determine how the left and right subarrays are created and sorted. Let's break them down in simple terms:

1. **left = [x for x in arr[:pivot\_index] + arr[pivot\_index+1:] if x <= pivot]** creates a list called **left**. This list will contain all the elements from the original array (**arr**) that are less than or equal to the chosen pivot element.
2. **right = [x for x in arr[:pivot\_index] + arr[pivot\_index+1:] if x > pivot]** creates another list called **right**. This list will contain all the elements from the original array that are greater than the chosen pivot element.
3. The idea here is to split the original array into two parts: **left**, which contains elements smaller or equal to the pivot, and **right**, which contains elements larger than the pivot.
4. After splitting the array, the code recursively applies the randomized Quick Sort algorithm to both the **left** and **right** subarrays.
5. The pivot element, which was chosen randomly, is placed in the middle of the sorted result.
6. Finally, the code returns the combined result of sorting the **left** subarray, the pivot element, and the sorted **right** subarray.