



DAA Viva



Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

Given above is the Problem statement for the lab practical in th subject of DAA. attached the Code for the implementation Explain the Code Line By line in detail.

```
public class DAA_1_FibonacciSeries {
    public static int recursive(int n) {
        if (n<=1) {return n;}
        return recursive(n-1)+recursive(n-2);
        //Time Complexity: O(2^n)
        //Space Complexity: O(n)
    }

    public static int iterative(int n) {
        int prev = 0;
        int cur = 1;
        int next = 0;
        for (int i = 2; i <= n; i++) {
            next = prev + cur;
            prev = cur;
            cur = next;
        }
        return cur;
        //Time Complexity: O(n)
        //Space Complexity: O(1)
    }

    public static void main(String[] args) {
        System.out.println(recursive(10));
        System.out.println(iterative(10));
    }
}
```



 **Memory updated**

Here's a detailed line-by-line explanation of the code you provided for calculating Fibonacci numbers using both recursive and iterative methods.

Code Explanation

```
java
```

```
public class DAA_1_FibonacciSeries {
```

- This declares a public class named `DAA_1_FibonacciSeries`. This is the main class that contains methods for calculating Fibonacci numbers.

Recursive Method

```
java
```

```
public static int recursive(int n) {
```

- Declares a `public` and `static` method named `recursive` that takes an integer `n` as input and returns an integer result.
- This method calculates the `n`-th Fibonacci number using recursion.

```
java
```

```
if (n <= 1) { return n; }
```

- This is a base case. If `n` is `0` or `1`, it directly returns `n` since the first two numbers in the Fibonacci sequence are defined as `0` and `1`.

```
java
```

```
return recursive(n - 1) + recursive(n - 2);
```

- For any other case (`n > 1`), the function calls itself with `(n-1)` and `(n-2)` and adds the results.
- This is based on the definition of Fibonacci numbers, where each number is the sum of the two preceding ones.

```
java
```

```
// Time Complexity: O(2^n)
// Space Complexity: O(n)
```

- **Time Complexity:** $O(2^n)$ because each call to `recursive` spawns two additional calls, resulting in exponential growth.
- **Space Complexity:** $O(n)$ due to the stack space needed for each recursive call.

Iterative Method

```
java
```

```
public static int iterative(int n) {
```

- Declares a `public` and `static` method named `iterative` that takes an integer `n` as input and returns an integer result.
- This method calculates the `n`-th Fibonacci number using an iterative approach.

```
java
```

```
int prev = 0;
int cur = 1;
int next = 0;
```

- Initializes three integer variables: `prev`, `cur`, and `next`.
 - `prev` holds the (n-2)-th Fibonacci number.
 - `cur` holds the (n-1)-th Fibonacci number.
 - `next` will store the current Fibonacci number.

```
java
```

```
for (int i = 2; i <= n; i++) {
```

- Starts a loop from `2` up to `n`, as the base cases for `0` and `1` are already covered by initialization.

```
java
```

```
next = prev + cur;
```

- Calculates the current Fibonacci number by adding `prev` and `cur`.

```
java
```

```
prev = cur;
cur = next;
```

- Updates `prev` to hold the previous `cur` value and `cur` to hold the `next` Fibonacci number.
- This shifts the sequence forward, preparing for the next iteration.

```
java
```

```
return cur;
```

- Returns the value of `cur`, which holds the `n`-th Fibonacci number after the loop ends.

```
java
```

```
// Time Complexity: O(n)
// Space Complexity: O(1)
```

- **Time Complexity:** $O(n)$ because it iterates n times.
- **Space Complexity:** $O(1)$ because it only uses a constant amount of extra space for variables.

Main Method

```
java
```

```
public static void main(String[] args) {
    System.out.println(recursive(10));
    System.out.println(iterative(10));
}
```

- The `main` method serves as the entry point of the program.
- It calls the `recursive` and `iterative` methods with `n = 10`, printing the 10th Fibonacci number using both approaches.
- `System.out.println` outputs the result to the console.

Summary of Analysis

- **Recursive Approach:**
 - Uses a tree-like structure, leading to exponential time complexity $O(2^n)$.
 - Requires $O(n)$ space due to the call stack.
- **Iterative Approach:**
 - Linear time complexity $O(n)$ as it iterates n times.
 - Constant space complexity $O(1)$ as it only uses a few variables.

This program demonstrates both recursive and iterative approaches to solving the Fibonacci sequence problem, highlighting their differences in time and space complexity.



Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.



Here are some commonly asked questions and suitable answers that you may encounter in a viva for this Fibonacci program:

1. What is a Fibonacci sequence?

- **Answer:** The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. Mathematically, it's defined as:
 - $F(0) = 0$
 - $F(1) = 1$
 - $F(n) = F(n - 1) + F(n - 2)$ for $n > 1$
-

2. What is recursion, and how does it work in this program?

- **Answer:** Recursion is a technique where a function calls itself to solve a smaller instance of the same problem. In this program, the `recursive` method calls itself twice, with `n-1` and `n-2`, until reaching the base cases $F(0)$ or $F(1)$. This approach works by building a tree-like structure where each call splits into two sub-calls, representing the Fibonacci formula.
-

3. Explain the time and space complexity of the recursive approach.

- **Answer:** The time complexity of the recursive approach is $O(2^n)$, because each call to the function generates two more calls, creating an exponential number of function calls as `n` increases. The space complexity is $O(n)$ due to the recursive call stack, which can grow up to `n` levels deep.
-

4. Why is the recursive method inefficient for large values of `n`?

- **Answer:** The recursive method is inefficient because it has exponential time complexity $O(2^n)$. This results in a large number of redundant calculations, where the same Fibonacci number is recalculated multiple times. For example, calculating `recursive(10)` requires recalculating many values like `recursive(2)` and `recursive(3)` repeatedly, leading to slower performance for large `n`.
-

5. Explain the iterative approach and its advantages.

- **Answer:** The iterative approach calculates Fibonacci numbers using a `for` loop, which updates two variables (holding the previous two Fibonacci numbers) to compute the next one. This approach has a linear time complexity $O(n)$ and constant space complexity $O(1)$. It's much more efficient than recursion for large `n`, as it avoids redundant calculations and does not use additional stack space.
-

6. What is the difference between time complexity and space complexity?

- **Answer:** Time complexity measures the amount of time an algorithm takes to run relative to the size of its input. Space complexity measures the amount of memory (or space) the algorithm uses during execution. For example, the recursive Fibonacci method has $O(2^n)$ time complexity and $O(n)$ space complexity, while the iterative method has $O(n)$ time complexity and $O(1)$ space complexity.
-

7. What are the base cases in the recursive Fibonacci method, and why are they necessary?

- **Answer:** The base cases in the recursive Fibonacci method are $F(0) = 0$ and $F(1) = 1$. These are necessary because they stop the recursion from going indefinitely and provide the initial known values of the Fibonacci sequence, allowing us to build the sequence for larger values of `n`.
-

8. Can you identify any potential drawbacks of recursion?

- **Answer:** Yes, recursion can lead to higher memory usage because each recursive call adds a new frame to the call stack, which can cause stack overflow errors for large input values. Additionally, recursion can be inefficient for problems with overlapping subproblems, like the Fibonacci sequence, because it may result in redundant calculations, increasing the time complexity.
-

9. How does memoization improve the recursive Fibonacci approach?

- **Answer:** Memoization is a technique where we store the results of function calls and reuse them instead of recalculating. In the recursive Fibonacci approach, memoization can save previously computed Fibonacci numbers in an array or map, drastically reducing the time complexity from $O(2^n)$ to $O(n)$ by avoiding redundant calculations.
-

10. What would happen if we try to compute a very large Fibonacci number using the recursive method?

- **Answer:** For very large values of `n`, the recursive method would take a long time due to its exponential time complexity. It could also result in a stack overflow because each recursive call consumes stack space, and the call stack would become very deep.
-

11. What is dynamic programming, and how does it relate to the iterative approach?

- **Answer:** Dynamic programming is a method used to solve problems by breaking them down into overlapping subproblems, solving each only once, and storing the results for reuse. The iterative Fibonacci approach can be considered a form of dynamic programming because it builds up the solution from the base cases without redundant calculations, using only the previous two values to find the next one.
-

12. Why is the space complexity $O(1)$ for the iterative method?

- **Answer:** The space complexity is $O(1)$ because the iterative method only uses a fixed number of variables (`prev`, `cur`, and `next`) regardless of the input size `n`. It doesn't require additional memory for an array or recursion stack.
-

13. Can the iterative method be modified to handle negative indices for Fibonacci numbers?

- **Answer:** Yes, the Fibonacci sequence can be extended to negative indices, resulting in a sequence known as the "negafibonacci." The formula for negative indices is $F(-n) = (-1)^{n+1} \cdot F(n)$. This could be implemented by adjusting the iterative method to calculate and handle negative `n` values accordingly.

These questions cover a wide range of topics around recursion, iteration, time and space complexity, and practical applications, which should help you in preparing for your viva.



Write a program to implement Huffman Encoding using a greedy strategy.

Given above is the Problem statement for the lab practical in th subject of DAA. attached the Code for the implementation Explain the Code Line By line in detail.

```
import java.util.Comparator;
import java.util.HashMap;
import java.util.PriorityQueue;
import java.util.Scanner;

class DAA_2_HuffmanEncoding {
    public static void printCode(HuffmanNode root, String s) {
        if (root.left == null && root.right == null
            && Character.isLetter(root.c)) {
            System.out.println(root.c + ":" + s);
            return;
        } else {
            printCode(root.left, s + "0");
            printCode(root.right, s + "1");
        }
    }
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the string to convert into Huffman Code");
    String s = sc.nextLine();
    HashMap<Character, Integer> hm = new HashMap<>();
    for (int i = 0; i < s.length(); i++) {
        hm.put(s.charAt(i), hm.getOrDefault(s.charAt(i), 0) + 1);
    }
    Character[] charArray = hm.keySet().toArray(new Character[0]);
    PriorityQueue<HuffmanNode> q = new PriorityQueue<>(charArray.length, new
MyComparator());
    for (Character character : charArray) {
        HuffmanNode hn = new HuffmanNode();
        hn.c = character;
        hn.data = hm.get(character);
        hn.left = null;
        hn.right = null;
    }
}
```

```

        q.add(hn);
    }
    HuffmanNode root = null;
    while (q.size() > 1) {
        HuffmanNode x = q.peek();
        q.poll();
        HuffmanNode y = q.peek();
        q.poll();
        HuffmanNode f = new HuffmanNode();
        f.data = x.data + y.data;
        f.c = '-';
        f.left = x;
        f.right = y;
        root = f;
        q.add(f);
    }
    printCode(root, "");
}
}

class HuffmanNode {
    int data;
    char c;
    HuffmanNode left;
    HuffmanNode right;
}

class MyComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y) {
        return x.data - y.data;
    }
}

```



Here's a line-by-line breakdown of the code for implementing Huffman Encoding using a greedy strategy in Java.

Code Explanation

java

```

import java.util.Comparator;
import java.util.HashMap;
import java.util.PriorityQueue;
import java.util.Scanner;

```

- **Imports:** The code imports necessary classes:
 - `Comparator` is used to define custom sorting for the `PriorityQueue`.

- `HashMap` is used to store character frequencies.
- `PriorityQueue` is used to build the Huffman Tree by always extracting nodes with the lowest frequency.
- `Scanner` is used to take input from the user.

```
java
```

```
class DAA_2_HuffmanEncoding {
    public static void printCode(HuffmanNode root, String s) {
        if (root.left == null && root.right == null
            && Character.isLetter(root.c)) {
            System.out.println(root.c + ":" + s);
            return;
        } else {
            printCode(root.left, s + "0");
            printCode(root.right, s + "1");
        }
    }
}
```

- **`printCode` Method:** This recursive function prints the Huffman code for each character.
 - **Parameters:**
 - `root`: The current node in the Huffman Tree.
 - `s`: The binary string representing the path from the root to the current node.
 - **Base Case:** If the node is a leaf (both left and right children are null) and contains a character (`Character.isLetter(root.c)`), it prints the character and its binary code (`s`).
 - **Recursive Case:** Calls itself on the left child, appending "0" to the code, and on the right child, appending "1".

```
java
```

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the string to convert into Huffman Code");
    String s = sc.nextLine();
    HashMap<Character, Integer> hm = new HashMap<>();
    for (int i = 0; i < s.length(); i++) {
        hm.put(s.charAt(i), hm.getOrDefault(s.charAt(i), 0) + 1);
    }
}
```

- **`main` Method:** The main entry point of the program.
 - Prompts the user to enter a string.
 - **Frequency Map (`hm`):** Uses a `HashMap` to calculate the frequency of each character in the input string. The `getOrDefault` method fetches the current count of each character, defaulting to 0 if the character isn't in the map, then increments it.

```
java
```

```
Character[] charArray = hm.keySet().toArray(new Character[0]);
PriorityQueue<HuffmanNode> q = new PriorityQueue<>(charArray.length, new MyCompara
tor());
for (Character character : charArray) {
    HuffmanNode hn = new HuffmanNode();
    hn.c = character;
    hn.data = hm.get(character);
}
```

```

        hn.left = null;
        hn.right = null;
        q.add(hn);
    }

```

- **Character Array:** Creates an array of unique characters in the input string.
- **Priority Queue (`q`):** Creates a priority queue of `HuffmanNode` objects, ordered by frequency using `MyComparator`.
- **Node Creation and Queue Population:**
 - For each character in the input, a `HuffmanNode` is created.
 - The character and its frequency are stored in the node, and the node is added to the priority queue.

java

```

HuffmanNode root = null;
while (q.size() > 1) {
    HuffmanNode x = q.peek();
    q.poll();
    HuffmanNode y = q.peek();
    q.poll();
    HuffmanNode f = new HuffmanNode();
    f.data = x.data + y.data;
    f.c = '-';
    f.left = x;
    f.right = y;
    root = f;
    q.add(f);
}

```

- **Building the Huffman Tree:**
 - The loop runs until only one node remains in the priority queue, which will be the root of the Huffman Tree.
 - Each time, the two nodes with the lowest frequencies are extracted from the queue (`x` and `y`).
 - A new node (`f`) is created with:
 - Frequency equal to the sum of `x` and `y`.
 - Character set as `'-` (a placeholder indicating it's not a leaf node).
 - `x` and `y` as the left and right children, respectively.
 - This new node is added back to the priority queue.

java

```

        printCode(root, "");
    }
}

```

- **Print Huffman Codes:** Calls `printCode` with the root node and an empty string, initiating the recursive function to print the Huffman codes for each character.

```
java
```

```
class HuffmanNode {
    int data;
    char c;
    HuffmanNode left;
    HuffmanNode right;
}
```

- **HuffmanNode Class:** Represents each node in the Huffman Tree.
 - `data`: Stores the frequency of the character.
 - `c`: Stores the character.
 - `left` and `right`: References to the left and right children in the tree.

```
java
```

```
class MyComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y) {
        return x.data - y.data;
    }
}
```

- **Comparator Class (`MyComparator`):** Implements `Comparator` to order nodes by frequency.
 - `compare`: Compares two `HuffmanNode` objects based on their `data` (frequency), ensuring that the priority queue orders nodes with lower frequencies first.

Analysis of the Code

- **Time Complexity:**
 - Constructing the frequency map takes $O(n)$, where n is the length of the input string.
 - Inserting nodes into the priority queue takes $O(m \log m)$, where m is the number of unique characters.
 - Building the Huffman Tree also takes $O(m \log m)$.
 - The overall time complexity is approximately $O(n + m \log m)$.
- **Space Complexity:**
 - The `HashMap` for storing frequencies requires $O(m)$ space.
 - The `PriorityQueue` also requires $O(m)$ space.
 - The Huffman Tree structure requires $O(m)$ space as well.
 - Thus, the space complexity is $O(m)$.



Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.



For a viva on this Huffman Encoding implementation practical, here are some questions that are likely to be asked, along with suitable answers:

1. What is Huffman Encoding, and why do we use it?

- **Answer:** Huffman Encoding is a lossless data compression algorithm. It reduces the size of data by encoding frequently occurring characters with shorter codes and less frequent characters with longer codes. This approach is based on the frequency of each character, allowing us to efficiently compress data without losing any information.

2. Explain the working of the Huffman Encoding algorithm.

- **Answer:** Huffman Encoding uses a greedy approach to build a binary tree, known as the Huffman Tree, based on the frequency of each character. The algorithm works as follows:
 1. Count the frequency of each character in the data.
 2. Insert each character as a node in a priority queue based on its frequency.
 3. While there's more than one node in the queue:
 - Remove the two nodes with the lowest frequency.
 - Create a new node with these two nodes as children and a frequency equal to their combined frequency.
 - Insert this new node back into the queue.
 4. Repeat until only one node remains. This node becomes the root of the Huffman Tree.
 5. Assign binary codes to each character by traversing the tree: '0' for left branches and '1' for right branches.

3. Why is Huffman Encoding considered a greedy algorithm?

- **Answer:** Huffman Encoding is considered a greedy algorithm because at each step, it picks the two nodes with the smallest frequency and combines them. This local optimization (combining the smallest nodes first) eventually leads to a globally optimal solution in terms of minimum encoding length.

4. What is the time complexity of Huffman Encoding?

- **Answer:** The time complexity of Huffman Encoding is $O(n + m \log m)$, where n is the length of the input string and m is the number of unique characters in the string. Constructing the frequency map takes $O(n)$, inserting nodes into the priority queue takes $O(m \log m)$, and building the Huffman Tree also takes $O(m \log m)$.

5. What is the difference between the left and right child in the Huffman Tree?

- **Answer:** In the Huffman Tree, the left child represents the binary '0', and the right child represents '1'. These binary values are used to construct the Huffman codes for each character based on the path from the root to the leaf node of that character.

6. How does Huffman Encoding ensure that it is a prefix-free code?

- **Answer:** Huffman Encoding generates prefix-free codes, meaning that no code is a prefix of another code. This is achieved because each character's code corresponds to a unique path from the root to a leaf node in the binary Huffman Tree. Since only leaf nodes contain characters, no code can be a prefix of another, ensuring that the encoded data can be uniquely decoded.

7. What are the space and time complexities of storing and generating the Huffman Tree?

- **Answer:** The space complexity for storing the Huffman Tree is $O(m)$, where m is the number of unique characters. This is because each character node is stored once. The time complexity for generating the Huffman Tree is $O(m \log m)$ due to the operations on the priority queue.

8. What are some applications of Huffman Encoding?

- **Answer:** Huffman Encoding is widely used in data compression algorithms such as ZIP files, JPEG image compression, and MP3 audio compression. It is also used in network protocols to reduce the amount of data transmitted, optimizing storage and transmission efficiency.

9. What would happen if all characters have the same frequency?

- **Answer:** If all characters have the same frequency, Huffman Encoding would still create a binary tree, but each character would end up with codes of similar length. The resulting Huffman Tree would be balanced, and while it would not achieve maximum compression, it would still encode the data with a unique prefix-free code.

10. What is the difference between Huffman Encoding and other compression algorithms?

- **Answer:** Huffman Encoding is a lossless compression algorithm, which means no data is lost during compression, unlike lossy algorithms such as JPEG. It is also character-based, meaning it compresses data by replacing individual characters with shorter codes. Other algorithms, like Run-Length Encoding, compress data by replacing repeated sequences, and algorithms like LZ77 are dictionary-based, which look for repeating patterns rather than individual character frequencies.

11. How does the priority queue help in building the Huffman Tree?

- **Answer:** The priority queue stores nodes in ascending order of frequency. By always removing the two nodes with the lowest frequency and combining them, we ensure that characters with lower frequencies end up deeper in the tree, which results in longer codes, while more frequent characters have shorter paths, leading to shorter codes. This organization is crucial to achieving efficient compression.

12. Explain the role of `Comparator` in this code.

- **Answer:** The `Comparator` is used to define a custom sorting order for the priority queue. In this case, `MyComparator` is implemented to compare `HuffmanNode` objects based on their frequency. This allows the priority queue to always have the lowest-frequency nodes at the top, ensuring that the tree is built optimally.

13. What modifications would you make to decode a Huffman-encoded message?

- **Answer:** To decode a Huffman-encoded message, we need to traverse the Huffman Tree. Starting from the root, for each '0' in the code, we move left, and for each '1', we move right. When we reach a leaf node, we output the character at that node and restart from the root for the next part of the message.

the next part of the encoded message. This process continues until the entire message is decoded.

14. Can Huffman Encoding be used to compress all types of data?

- **Answer:** Huffman Encoding is effective for compressing data with a non-uniform frequency distribution, where some characters appear more frequently than others. However, for data with uniform or random distributions, Huffman Encoding may not yield significant compression. In some cases, other compression methods might be more efficient.

15. What are the limitations of Huffman Encoding?

- **Answer:** Some limitations of Huffman Encoding include:
 - It may not provide significant compression for data with uniform character frequencies.
 - It requires building a frequency table and Huffman Tree, which can add overhead.
 - It does not work as well on small datasets or data with a high degree of randomness.
 - It is less effective for files with few unique characters, as there isn't much opportunity for optimization.

These questions cover a wide range of theoretical and practical aspects of Huffman Encoding and demonstrate a solid understanding of the code and its applications.



Write a program to solve a fractional Knapsack problem using a greedy method.

Given above is the Problem statement for the lab practical in th subject of DAA. attached the Code for the implementation Explain the Code Line By line in detail.

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

class WeightNode {
    int weight;
    int profit;

    public WeightNode(int weight, int profit) {
        this.weight = weight;
        this.profit = profit;
    }
}

class DAA_3_FractionalKnapsack {

    public static double profitMaximizer(PriorityQueue<WeightNode> pq, int capacity) {
        double profit = 0;
        while (!pq.isEmpty() && capacity > 0) {
            WeightNode w = pq.poll();
```

```

        if (capacity > w.weight) {
            capacity -= w.weight;
            profit += w.profit;
        }
        else {
            profit += w.profit*((double) capacity /w.weight);
        }
    }
    return profit;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number of items");
    int n = sc.nextInt();
    System.out.println("Enter the weight and profit seperated by space");
    PriorityQueue<WeightNode> pq = new PriorityQueue<>((WeightNode x, WeightNode y) -> {
        double X = (double) x.profit / x.weight;
        double Y = (double) y.profit / y.weight;
        return X < Y ? 1 : -1;
    });
    for (int i = 0; i < n; i++) {
        pq.add(new WeightNode(sc.nextInt(), sc.nextInt()));
    }
    System.out.println("Enter The Capacity of Knapsack");
    int capacity = sc.nextInt();
    System.out.println(profitMaximizer(pq, capacity));
}
}

```



Here's a line-by-line explanation of the code for solving the **Fractional Knapsack** problem using a greedy algorithm.

Code Explanation

java

```

import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

```

- **Imports:** The code imports `Comparator` for custom sorting, `PriorityQueue` for storing and sorting items by profit-to-weight ratio, and `Scanner` for user input.

Class Definitions

```
java
```

```
class WeightNode {
    int weight;
    int profit;

    public WeightNode(int weight, int profit) {
        this.weight = weight;
        this.profit = profit;
    }
}
```

- **WeightNode Class:** This class represents each item, with `weight` and `profit` properties.
- **Constructor:** The constructor `WeightNode(int weight, int profit)` initializes each item's weight and profit.

Fractional Knapsack Implementation

```
java
```

```
class DAA_3_FractionalKnapsack {
```

- **Main Class:** This is the main class, containing the logic for solving the Fractional Knapsack problem.

`profitMaximizer` Method

```
java
```

```
public static double profitMaximizer(PriorityQueue<WeightNode> pq, int capacity) {
    double profit = 0;
```

- **profitMaximizer Method:** This method calculates the maximum profit possible within the given knapsack capacity.
- **`profit` Variable:** Initializes the total profit as `0`. This variable accumulates profit as items are added to the knapsack.

```
java
```

```
while (!pq.isEmpty() && capacity > 0) {
    WeightNode w = pq.poll();
```

- **Loop:** Iterates as long as there are items in the priority queue and the knapsack has remaining capacity.
- **`pq.poll()`:** Retrieves and removes the item with the highest profit-to-weight ratio from the priority queue.


```
java
```

```
        if (capacity > w.weight) {
            capacity -= w.weight;
            profit += w.profit;
        }
```

- **Full Item Inclusion:** If the knapsack capacity is greater than the item's weight, the entire item is added to the knapsack.
 - `capacity -= w.weight`: Reduces the knapsack capacity by the weight of the item.
 - `profit += w.profit`: Increases the total profit by the item's profit.

```
java
```

```
        else {
            profit += w.profit * ((double) capacity / w.weight);
        }
    }
```

- **Fractional Inclusion:** If the knapsack capacity is less than the item's weight, only a fraction of the item is added.
 - **Fractional Profit:** Calculates the fraction of the profit by multiplying the item's profit by the fraction of its weight that fits into the remaining capacity.

```
java
```

```
    }    return profit;
}
```

- **Return Profit:** The method returns the total profit accumulated.

Main Method

```
java
```

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number of items");
    int n = sc.nextInt();
}
```

- **Input - Number of Items:** The user is prompted to enter the number of items, which is stored in `n`.

```
java
```

```
    System.out.println("Enter the weight and profit separated by space");
    PriorityQueue<WeightNode> pq = new PriorityQueue<>((WeightNode x, WeightNode y) ->
    {
        double X = (double) x.profit / x.weight;
    });
```

```
double Y = (double) y.profit / y.weight;
return X < Y ? 1 : -1;
});
```

- **Input - Weights and Profits:** A prompt to enter each item's weight and profit.
- **Priority Queue Initialization:** Initializes a priority queue with a custom comparator.
 - **Comparator:** Orders items based on their profit-to-weight ratio (descending order). Items with a higher ratio are prioritized for maximum profit.

java

```
for (int i = 0; i < n; i++) {
    pq.add(new WeightNode(sc.nextInt(), sc.nextInt()));
}
```

- **Loop to Add Items:** Adds each item to the priority queue by reading the weight and profit values from user input.

java

```
System.out.println("Enter The Capacity of Knapsack");
int capacity = sc.nextInt();
```

- **Input - Knapsack Capacity:** The user enters the maximum capacity of the knapsack, stored in `capacity`.

java

```
        System.out.println(profitMaximizer(pq, capacity));
    }
}
```

- **Calculate and Print Profit:** Calls `profitMaximizer` to calculate the maximum profit and prints the result.

Explanation of Key Points

- **Greedy Method:** The algorithm uses a greedy strategy by prioritizing items with the highest profit-to-weight ratio, ensuring optimal selection.
 - **Fractional Selection:** If an item cannot be fully included, the algorithm adds a fractional portion based on remaining knapsack capacity.
-

Complexity Analysis

1. Time Complexity: $O(n \log n)$

- Sorting items by their profit-to-weight ratio in the priority queue requires $O(n \log n)$.
- Processing items from the priority queue has $O(n)$ complexity.

2. Space Complexity: $O(n)$

- The space complexity arises from storing items in the priority queue.



Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.



For a viva on implementing the Fractional Knapsack problem using a greedy approach, here are some questions you might expect along with detailed answers:

1. Explain the Fractional Knapsack Problem.

- **Answer:** The Fractional Knapsack Problem is a variant of the classic Knapsack Problem where items can be broken into fractions. In this problem, each item has a weight and a profit, and the goal is to maximize the profit by selecting items or fractions of items that can fit into a knapsack with a given capacity. The greedy approach is typically used to solve this problem by prioritizing items with the highest profit-to-weight ratio.

2. Why is the greedy approach suitable for the Fractional Knapsack Problem?

- **Answer:** The greedy approach is suitable because it maximizes profit by prioritizing items based on their profit-to-weight ratio. By choosing items with the highest ratio first, we ensure that the knapsack's capacity is used most profitably. The Fractional Knapsack Problem has the **Optimal Substructure** property, meaning that an optimal solution can be constructed by selecting optimal solutions to subproblems, which is why a greedy algorithm is effective.

3. How does the Fractional Knapsack Problem differ from the 0/1 Knapsack Problem?

- **Answer:** In the **Fractional Knapsack Problem**, you can take fractions of an item, allowing more flexibility in filling the knapsack optimally. In contrast, the **0/1 Knapsack Problem** does not allow breaking items into smaller parts—each item is either taken fully or not taken at all. The 0/1 Knapsack Problem typically requires a **dynamic programming** approach to find the optimal solution, while the Fractional Knapsack Problem can be solved greedily.

4. Explain the role of the priority queue in your implementation.

- **Answer:** The priority queue in this implementation is used to automatically sort and retrieve items based on their profit-to-weight ratio, which is computed in the comparator. Items with higher ratios are prioritized at the top of the queue, ensuring that the greedy approach can access the most profitable item first. This ordering is essential to ensure that the solution maximizes profit efficiently.

5. What is the time complexity of this algorithm, and why?

- **Answer:** The time complexity is $O(n \log n)$. This complexity arises from inserting n items into a priority queue, which takes $O(\log n)$ time for each item due to the sorting mechanism in the queue. Thus, inserting all items has a time complexity of $O(n \log n)$. Additionally, processing items from the queue in the main loop takes $O(n)$ time, but this is dominated by the sorting time.

6. What is the space complexity of this algorithm, and why?

- **Answer:** The space complexity is $O(n)$, as we store n items in the priority queue. Additionally, each item is represented by a `WeightNode` object containing its weight and profit, so we need space for n `WeightNode` objects.

7. How does the comparator work in the priority queue?

- **Answer:** The comparator orders items by their profit-to-weight ratio in descending order. For each item, the ratio is calculated as `profit / weight`. The comparator then compares these ratios and orders items so that those with higher ratios are retrieved first. This ordering allows the greedy method to choose the most profitable items efficiently.

8. Why do we use fractional values in the solution?

- **Answer:** We use fractional values because, in some cases, the remaining knapsack capacity may be less than the weight of an item. To maximize profit, we can take only a portion of such items rather than leaving them out. This fraction of the item's weight (proportional to the remaining capacity) is added to the knapsack to maximize the profit without exceeding the capacity.

9. What would happen if we removed the priority queue and used an array?

- **Answer:** If we used an array, we would need to sort it by profit-to-weight ratio first, which would still require $O(n \log n)$ time complexity. However, using a priority queue allows us to retrieve the maximum ratio element in constant time after each item is processed, making the approach slightly more efficient and better suited for scenarios where we repeatedly access the highest priority items.

10. Can the Fractional Knapsack Problem be solved using dynamic programming? Why or why not?

- **Answer:** No, dynamic programming is generally not used for the Fractional Knapsack Problem because the problem can be solved optimally with a greedy approach. Dynamic programming is typically reserved for problems like the 0/1 Knapsack Problem, where items cannot be split, and the problem requires exploring multiple combinations to find the optimal solution.

11. Explain the process of calculating the fractional profit in your code.

- **Answer:** If an item's weight exceeds the remaining knapsack capacity, we calculate the profit for the fraction of the item that fits. The fractional profit is calculated by multiplying the item's total profit by the ratio of the remaining capacity to the item's weight: `profit += w.profit * (capacity / w.weight)`. This allows us to add a proportional amount of profit for the fraction of the item included.

12. What are some real-life applications of the Fractional Knapsack Problem?

- **Answer:** Real-life applications include resource allocation where resources are divisible, such as budget allocation, where parts of a budget are allocated to maximize returns. It's also used in logistics, where cargo can be divided and loaded based on value density, or in financial investments, where assets are allocated based on return ratios.

13. What will the output be if the capacity of the knapsack is zero?

- **Answer:** If the knapsack capacity is zero, the output will be `0.0`, as no items can be added to the knapsack, resulting in no profit.

14. What are the limitations of the Fractional Knapsack Problem solution?

- **Answer:** One limitation is that it assumes items are divisible, which may not be practical in real-life scenarios where items must be taken in whole units (as in the 0/1 Knapsack Problem). Also, it does not consider constraints other than weight, such as volume or other dimensions, which might be relevant in complex packing problems.

These questions cover core concepts, implementation details, theoretical aspects, and practical applications, preparing you comprehensively for a viva on the Fractional Knapsack Problem.



Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Given above is the Problem statement for the lab practical in th subject of DAA. attached the Code for the implementation Explain the Code Line By line in detail.

```
public class DAA_4_01Knapsack {
    static int knapSack(int W, int wt[], int val[], int n)
    {
        // Base Case
        if (n == 0 || W == 0)
            return 0;

        // If weight of the nth item is
        // more than Knapsack capacity W,
        // then this item cannot be included
        // in the optimal solution
        if (wt[n - 1] > W)
            return knapSack(W, wt, val, n - 1);

        // Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        else
            return Math.max(knapSack(W, wt, val, n - 1),
                           val[n - 1] + knapSack(W - wt[n-1], wt, val, n-1));
    }
}
```

```
// Driver code
public static void main(String args[])
{
    int profit[] = new int[] { 60, 100, 120 };
    int weight[] = new int[] { 10, 20, 30 };
    int W = 50;
    int n = profit.length;
    System.out.println(knapSack(W, weight, profit, n));
}
}
```



Here's a detailed explanation of the code implementing the 0-1 Knapsack Problem using a recursive approach with dynamic programming principles:

java

```
public class DAA_4_01Knapsack {
    static int knapSack(int W, int wt[], int val[], int n) {
        // Base Case
        if (n == 0 || W == 0)
            return 0;
    }
```

1. Function Definition:

- `knapSack(int W, int wt[], int val[], int n)` is a recursive function that solves the 0-1 Knapsack problem.
- Parameters:
 - `W` is the remaining weight capacity of the knapsack.
 - `wt[]` is an array containing weights of each item.
 - `val[]` is an array containing the profit (or value) of each item.
 - `n` is the number of items remaining for consideration.

2. Base Case:

- If there are no items left to consider (`n == 0`) or the knapsack capacity (`W`) is zero, the function returns `0`, meaning no profit can be made in this case.

java

```
// If weight of the nth item is
// more than Knapsack capacity W,
// then this item cannot be included
// in the optimal solution
if (wt[n - 1] > W)
    return knapSack(W, wt, val, n - 1);
```

3. Check Item Weight Against Capacity:

- If the weight of the last item (`wt[n - 1]`) exceeds the current knapsack capacity (`W`), we cannot include this item. Therefore, the function recursively calls `knapSack(W, wt, val, n - 1)`, excluding the current item from consideration.

```
java
```

```

        // Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
    else
        return Math.max(knapSack(W, wt, val, n - 1),
                        val[n - 1] + knapSack(W - wt[n-1], wt, val, n-1));
    }

```

4. Include or Exclude the Current Item:

- If the current item can fit in the knapsack (its weight is less than or equal to `W`), we have two choices:
 - **Exclude the item:** Call `knapSack(W, wt, val, n - 1)` to find the maximum profit without including the current item.
 - **Include the item:** Add the profit of the current item (`val[n - 1]`) and recursively call `knapSack(W - wt[n - 1], wt, val, n - 1)` to compute the remaining profit with reduced capacity.
- The `Math.max()` function is used to choose the option that provides the maximum profit.

```
java
```

```

// Driver code
public static void main(String args[]) {
    int profit[] = new int[] { 60, 100, 120 };
    int weight[] = new int[] { 10, 20, 30 };
    int W = 50;
    int n = profit.length;
    System.out.println(knapSack(W, weight, profit, n));
}

```

5. Driver Code (main method):

- The `main` method initializes:
 - `profit[]`: An array of profit values for each item.
 - `weight[]`: An array of weight values for each item.
 - `W`: The maximum capacity of the knapsack.
 - `n`: The total number of items available.
- The `knapSack` function is called with these values, and the maximum achievable profit is printed.

Example Walkthrough

1. Suppose `W = 50`, `profit = [60, 100, 120]`, and `weight = [10, 20, 30]`.
2. The recursive calls explore whether to include or exclude each item, ultimately returning the maximum profit achievable without exceeding `W`.

Complexity Analysis

1. **Time Complexity:** The recursive approach has an exponential time complexity of $O(2^n)$, where `n` is the number of items, as each item has two possibilities (include or exclude).
2. **Space Complexity:** Since each recursive call goes down by one item, the space complexity is $O(n)$, corresponding to the maximum depth of the recursive stack.

This recursive solution can be optimized by applying **memoization** or using **dynamic programming with tabulation** to reduce time complexity to $O(n \cdot W)$.



Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.



Here are some likely questions that could be asked in a viva based on the 0-1 Knapsack Problem code provided, along with suitable answers:

1. What is the 0-1 Knapsack Problem?

- The 0-1 Knapsack Problem is a combinatorial optimization problem where we are given items with specific weights and profits. Our goal is to select items to maximize the total profit without exceeding a given weight capacity. In the 0-1 version, each item can be either included in the knapsack or excluded (hence "0-1").

2. What does the term "0-1" mean in this context?

- The term "0-1" means that for each item, we have only two choices: either to include it in the knapsack or to exclude it. We cannot include fractional parts of an item, which would be the case in a "Fractional Knapsack" problem.

3. Why did you use recursion in this code?

- We used recursion because it naturally fits the problem structure where each item has two options: include or exclude. Recursive calls help to explore both possibilities and compute the maximum profit by considering all valid combinations of items.

4. What is the base case in this recursive function?

- The base case is when there are no more items to consider ($n == 0$) or when the knapsack capacity is zero ($w == 0$). In these cases, the maximum profit is zero since no items can be selected or the capacity is insufficient to carry any items.

5. What are the time and space complexities of this approach?

- **Time Complexity:** The time complexity is $O(2^n)$ because for each item, there are two choices, leading to an exponential growth of combinations.
- **Space Complexity:** The space complexity is $O(n)$ due to the maximum depth of the recursive call stack.

6. How can this solution be optimized?

- This solution can be optimized using **dynamic programming (DP)** with memoization or tabulation. By storing already computed results for specific capacities and item counts, we avoid

redundant calculations. Using DP, the time complexity can be reduced to $O(n \cdot W)$.

7. What is memoization, and how would it help here?

- **Memoization** is a technique used to store results of expensive function calls for specific parameters so that these results can be reused in future calls. In the 0-1 Knapsack problem, it helps avoid recomputation by storing results for each unique subproblem defined by the current item and capacity.

8. Explain the `Math.max()` function usage in the code.

- The `Math.max()` function is used to determine the maximum profit between two choices: excluding the current item or including it. This comparison ensures that we select the option that maximizes profit without exceeding the knapsack capacity.

9. What is the difference between the 0-1 Knapsack and the Fractional Knapsack problem?

- In the **0-1 Knapsack problem**, each item can either be included or excluded entirely, with no partial items allowed. In the **Fractional Knapsack problem**, we are allowed to take fractions of items, which is usually solved with a greedy algorithm based on the item's profit-to-weight ratio.

10. What is the purpose of the `knapsack()` function parameters: `W`, `wt[]`, `val[]`, and `n`?

- `W` represents the maximum weight capacity of the knapsack.
- `wt[]` is an array containing the weights of the items.
- `val[]` is an array containing the profits (values) of the items.
- `n` represents the number of items still being considered in the recursive process.

11. What is a greedy strategy, and why doesn't it work for the 0-1 Knapsack problem?

- A **greedy strategy** is an approach that makes the best immediate choice at each step, such as choosing items based on the highest profit-to-weight ratio. However, it does not work optimally for the 0-1 Knapsack because taking items solely based on this ratio may lead to a suboptimal solution when items are indivisible.

12. Can you explain the difference between top-down and bottom-up DP?

- **Top-Down DP (Memoization)**: Uses recursion and stores intermediate results to avoid redundant calculations.
- **Bottom-Up DP (Tabulation)**: Builds solutions iteratively by solving all subproblems up to the desired result, typically filling out a table.

13. If `W = 50` and the item weights are `[10, 20, 30]`, with profits `[60, 100, 120]`, what would the maximum profit be?

- For `W = 50`, with `weights = [10, 20, 30]` and `profits = [60, 100, 120]`, the maximum achievable profit is **220**. This comes from including the items with weights 20 and 30, which provide the highest combined profit under the weight constraint.

14. What is the difference between recursive and dynamic programming solutions for this problem?

- The recursive solution explores all possible combinations and recalculates results for overlapping subproblems, leading to an exponential time complexity. In contrast, dynamic programming stores results of subproblems, reducing time complexity by avoiding redundant computations.

15. Could you write the iterative (bottom-up) dynamic programming approach for this problem?

- The bottom-up approach uses a 2D DP array where each cell `dp[i][j]` represents the maximum profit achievable with `i` items and a knapsack capacity of `j`. We can construct this table iteratively to obtain the final solution, which optimizes both time and space complexity.

These questions cover the theoretical understanding of the 0-1 Knapsack problem, details of the recursive approach, and comparisons with alternative methods.



Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.

Given above is the Problem statement for the lab practical in th subject of DAA. attached the Code for the implementation Explain the Code Line By line in detail.

```
import java.util.Scanner;

public class DAA_5_NQueens {
    private int n;
    private int[][] board;

    public DAA_5_NQueens(int n) {
        this.n = n;
        board = new int[n][n];
    }

    // Function to check if placing queen at board[row][col] is safe
    private boolean isSafe(int row, int col) {
        // Check this row on the left side
        for (int i = 0; i < col; i++) {
            if (board[row][i] == 1) {
                return false;
            }
        }

        // Check upper diagonal on the left side
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }
    }
}
```

```

    }
}

// Check lower diagonal on the left side
for (int i = row, j = col; j >= 0 && i < n; i++, j--) {
    if (board[i][j] == 1) {
        return false;
    }
}

return true;
}

// Recursive utility function to solve N-Queens problem
private boolean solveNQueens(int col) {
    // Base case: If all queens are placed
    if (col >= n) {
        return true;
    }

    // Try placing a queen in each row for this column
    for (int i = 0; i < n; i++) {
        // Check if queen can be placed on board[i][col]
        if (isSafe(i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place the rest of the queens
            if (solveNQueens(col + 1)) {
                return true;
            }

            // Backtrack if placing queen here doesn't lead to a solution
            board[i][col] = 0;
        }
    }

    // If no placement is possible, return false
    return false;
}

// Function to place the first queen and solve for remaining queens
public boolean placeQueens() {
    // Place the first queen at board[0][0] (can be modified as needed)
    Scanner scanner = new Scanner(System.in);
    System.out.println("Where Would You Like To place Your First Queen in First Column?");
    board[scanner.nextInt()][0] = 1;

    // Solve for the remaining queens starting from column 1
    return solveNQueens(1);
}

```

```
// Function to print the n-Queens matrix
public void printBoard() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.print(board[i][j] + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter The Size of Board");
    int n = scanner.nextInt(); // You can set n to any desired size
    DAA_5_NQueens nQueens = new DAA_5_NQueens(n);

    // Start placing queens and print the solution if it exists
    if (nQueens.placeQueens()) {
        System.out.println("Solution found:");
        nQueens.printBoard();
    } else {
        System.out.println("No solution exists for " + n + " queens.");
    }
}
}
```



Let's go through the code line by line and understand how it implements the solution for the N-Queens problem using backtracking:

Code Explanation

java

```
import java.util.Scanner;
```

- This imports the `Scanner` class from Java's standard library, which will be used for user input.

java

```
public class DAA_5_NQueens {
    private int n;
    private int[][] board;
```

- The `DAA_5_NQueens` class contains two fields:
 - `n`: the size of the N x N chessboard.

- `board`: a 2D array representing the chessboard where `1` indicates the presence of a queen, and `0` indicates an empty cell.

```
java
```

```
public DAA_5_NQueens(int n) {
    this.n = n;
    board = new int[n][n];
}
```

- This constructor initializes the `n` (size of the board) and creates an `n x n` matrix (`board`) initialized to `0` (no queens placed).

Function to Check Safety

```
java
```

```
private boolean isSafe(int row, int col) {
```

- `isSafe` checks if it is safe to place a queen at position `(row, col)` on the board.

```
java
```

```
// Check this row on the left side
for (int i = 0; i < col; i++) {
    if (board[row][i] == 1) {
        return false;
    }
}
```

- This loop checks if there is any queen on the same `row` to the left of the `(row, col)` position. If it finds a queen (`1`), it returns `false`.

```
java
```

```
// Check upper diagonal on the left side
for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
    if (board[i][j] == 1) {
        return false;
    }
}
```

- This loop checks the upper-left diagonal for any queens. If it finds a queen, it returns `false`.

```
java
```

```
// Check lower diagonal on the left side
for (int i = row, j = col; j >= 0 && i < n; i++, j--) {
    if (board[i][j] == 1) {
        return false;
    }
}
```

```
    }  
}
```

- This loop checks the lower-left diagonal for any queens. If it finds a queen, it returns `false`.

```
java
```

```
    return true;  
}
```

- If no queens are found in the row, upper-left diagonal, or lower-left diagonal, the function returns `true`, indicating it is safe to place a queen at `(row, col)`.

Recursive Function to Solve N-Queens

```
java
```

```
private boolean solveNQueens(int col) {
```

- This recursive function attempts to place queens in each column starting from `col`.

```
java
```

```
    // Base case: If all queens are placed  
    if (col >= n) {  
        return true;  
    }
```

- This is the base case of the recursion: if `col` reaches `n`, all queens have been successfully placed, and the function returns `true`.

```
java
```

```
    // Try placing a queen in each row for this column  
    for (int i = 0; i < n; i++) {
```

- This loop iterates through each row `i` of the current column `col`, trying to place a queen in each position `(i, col)`.

```
java
```

```
        // Check if queen can be placed on board[i][col]  
        if (isSafe(i, col)) {
```

- The function checks if it is safe to place a queen at `(i, col)` by calling `isSafe()`.

```
java
```

```
// Place this queen in board[i][col]
board[i][col] = 1;
```

- If it is safe, the queen is placed at `(i, col)` by setting `board[i][col]` to `1`.

```
java
```

```
// Recur to place the rest of the queens
if (solveNQueens(col + 1)) {
    return true;
}
```

- After placing the queen, the function recursively calls itself to try placing queens in the next column `(col + 1)`.
- If this recursive call returns `true`, it indicates that a valid solution has been found, and the function returns `true`.

```
java
```

```
// Backtrack if placing queen here doesn't lead to a solution
board[i][col] = 0;
    }
}
```

- If placing a queen at `(i, col)` and subsequent queens did not lead to a solution, the queen is removed from `(i, col)` (backtracking) by setting `board[i][col]` back to `0`.

```
java
```

```
// If no placement is possible, return false
return false;
}
```

- If no position in the current column `col` leads to a valid solution, the function returns `false`, indicating that no solution exists from this configuration.

Function to Place the First Queen

```
java
```

```
public boolean placeQueens() {
    // Place the first queen at board[0][0] (can be modified as needed)
    Scanner scanner = new Scanner(System.in);
    System.out.println("Where Would You Like To place Your First Queen in First Column?");
    board[scanner.nextInt()][0] = 1;
```

- This function initiates the placement of queens.
- It first prompts the user to input the row index for the first queen in the first column (column `0`).
- The first queen is placed in the user-specified row of the first column by setting `board[row][0]` to `1`.

```
java
```

```
    // Solve for the remaining queens starting from column 1
    return solveNQueens(1);
}
```

- It then calls `solveNQueens(1)` to place the remaining queens starting from the second column.

Function to Print the Board

```
java
```

```
public void printBoard() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.print(board[i][j] + " ");
        }
        System.out.println();
    }
}
```

- `printBoard()` prints the `n x n` board matrix to the console, where `1` represents a queen and `0` an empty space.

Main Method

```
java
```

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter The Size of Board");
    int n = scanner.nextInt(); // You can set n to any desired size
    DAA_5_NQueens nQueens = new DAA_5_NQueens(n);
}
```

- The `main` method begins by creating a `Scanner` instance for user input.
- It prompts the user to enter the size of the chessboard (`n`).
- It then creates an instance of `DAA_5_NQueens` with the specified size `n`.

```
java
```

```
// Start placing queens and print the solution if it exists
if (nQueens.placeQueens()) {
    System.out.println("Solution found:");
    nQueens.printBoard();
} else {
    System.out.println("No solution exists for " + n + " queens.");
}
```



```
}  
}
```

- The program calls `placeQueens()` to start solving the N-Queens problem.
- If a solution is found, it prints "Solution found:" and displays the board.
- If no solution exists, it outputs a message indicating that.

Summary

This program uses **backtracking** to solve the N-Queens problem:

1. The user specifies where to place the first queen.
2. The program then uses recursion to place additional queens while checking for conflicts.
3. If a conflict occurs, the program backtracks and tries a new configuration.
4. The process continues until all queens are safely placed or all configurations are exhausted.



Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.



For this practical on the N-Queens problem, here are some likely viva questions and detailed answers that could help prepare for the session:

1. What is the N-Queens Problem?

- **Answer:** The N-Queens problem is a classic backtracking problem where the objective is to place N queens on an N x N chessboard so that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal.

2. Explain the concept of backtracking and how it is used here.

- **Answer:** Backtracking is a problem-solving approach that incrementally builds a solution and, if it realizes that the current path will not lead to a solution, it "backtracks" to the previous step and tries a different option. In the N-Queens problem, we place queens one by one in each column. If placing a queen in a particular row leads to a conflict in the future columns, we backtrack by removing that queen and trying the next row in that column.

3. What does the `isSafe` method do in the code?

- **Answer:** The `isSafe` method checks if placing a queen at a specific `(row, col)` position is safe. It verifies that there is no other queen in the same row, upper-left diagonal, or lower-left diagonal. If there is any queen in these positions, the function returns `false`, indicating it's not safe to place a queen at `(row, col)`.

4. Why do we only check the left side of the board in `isSafe``?

- **Answer:** Since we place queens one column at a time from left to right, we only need to check previously placed queens in columns to the left of the current position. This optimization reduces unnecessary checks, as no queens will be placed in future columns that could conflict with the current column.
-

5. What is the purpose of backtracking in the `solveNQueens`` method?

- **Answer:** The `solveNQueens`` method uses backtracking to explore all possible placements of queens on the board. If a queen's placement leads to a conflict in future columns, we backtrack by removing the queen and trying the next possible row in the current column. This process continues recursively until a solution is found or all possibilities are exhausted.
-

6. Explain the recursive base case in the `solveNQueens`` function.

- **Answer:** The base case for `solveNQueens`` is `if (col >= n)``. This condition checks if we have placed queens in all columns, meaning we have successfully placed all N queens on the board without conflicts. If this is true, the function returns `true``, indicating that a solution has been found.
-

7. Why is there a need for the `placeQueens`` function, and what does it do?

- **Answer:** The `placeQueens`` function allows the user to specify the position of the first queen in the first column, which is an extra feature in this implementation. After placing the first queen, it calls `solveNQueens(1)`` to solve the problem for the remaining columns. This function introduces user input into the solution, making it slightly more interactive.
-

8. What will happen if there is no solution for a particular board size?

- **Answer:** If there is no possible solution to place N queens on the board without conflicts, `solveNQueens`` will ultimately return `false``. In this case, the `placeQueens`` method will print "No solution exists for n queens" to inform the user that it couldn't find a solution.
-

9. Can the N-Queens problem have multiple solutions?

- **Answer:** Yes, the N-Queens problem can have multiple solutions. This implementation finds only one solution due to the early return in `solveNQueens`` after the first valid configuration. To find all solutions, we would remove the `return true`` statement after a solution is found and allow the function to continue searching.
-

10. What is the time complexity of this solution?

- **Answer:** The time complexity of the N-Queens problem is **$O(N!)$** , where N is the number of queens (or the board size). This is because each queen has N possible rows to choose from initially, and each placement reduces the number of valid options for subsequent queens.
-

11. Why do we use a 2D array to represent the chessboard?

- **Answer:** We use a 2D array (`int[][] board`) to represent the chessboard so that each element corresponds to a specific cell on the board. Each cell can either be `0` (no queen) or `1` (queen present). This makes it easy to track queen positions and check for conflicts.

12. How would you modify the code to allow the user to choose a different position for the first queen?

- **Answer:** The code already prompts the user to select the row for the first queen in the first column. However, to allow the user to select both the row and column, we could modify the code in `placeQueens` to take inputs for both row and column positions and then place the first queen accordingly.

13. What are some practical applications of the N-Queens problem?

- **Answer:** The N-Queens problem and its solution using backtracking are primarily used for educational purposes to teach constraint satisfaction problems and backtracking algorithms. However, backtracking itself is used in various real-world applications, such as solving puzzles (Sudoku), resource allocation problems, scheduling tasks, and in optimization problems like pathfinding.

14. How does this algorithm handle the constraints of not placing queens in the same row, column, or diagonal?

- **Answer:** The algorithm ensures that no two queens share the same row, column, or diagonal by:
 - Placing one queen per column during each recursive call.
 - Using the `isSafe` method to check for other queens in the same row and diagonals to the left of the current position.
- If all these checks pass, a queen can be placed in the current cell.

15. If you encounter a memory constraint, how could you optimize the board representation?

- **Answer:** To save memory, instead of using a 2D array, we could use three separate 1D arrays:
 - `rows[i]` to check if a queen is present in row `i`.
 - `upperDiagonal[i]` to check the upper diagonal conflicts.
 - `lowerDiagonal[i]` to check the lower diagonal conflicts.
- This approach reduces space complexity and optimizes checks for queen placements.

16. What would be the output of the code if the user inputs `n = 3`?

- **Answer:** For `n = 3`, there is no solution to the 3-Queens problem, so the output would be:

```
arduino
```

```
No solution exists for 3 queens.
```

17. Explain why the N-Queens problem does not have solutions for $n = 2$ or $n = 3$.

- **Answer:** For $n = 2$ and $n = 3$, it is impossible to place the queens without conflicts due to the limited board size. The configurations in these board sizes do not allow enough space for queens to avoid each other in all directions, so there are no valid solutions for these cases.

These questions and answers should help cover fundamental concepts and logic behind the code, as well as some additional insights that may be useful during a viva session.



Write a program for analysis of quick sort by using deterministic and randomized variant.

Given above is the Problem statement for the lab practical in th subject of DAA. attached the Code for the implementation Explain the Code Line By line in detail.

```
import java.util.Arrays;
import java.util.Random;

public class DAA_6_QuickSort {
    // Function to perform Deterministic QuickSort
    public static void deterministicQuickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pivotIndex = deterministicPartition(arr, low, high);
            deterministicQuickSort(arr, low, pivotIndex - 1);
            deterministicQuickSort(arr, pivotIndex + 1, high);
        }
    }

    // Partition function for Deterministic QuickSort
    private static int deterministicPartition(int[] arr, int low, int high) {
        int pivot = arr[high]; // using the last element as pivot
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return i + 1;
    }
}
```

```

}

// Function to perform Randomized QuickSort
public static void randomizedQuickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivotIndex = randomizedPartition(arr, low, high);
        randomizedQuickSort(arr, low, pivotIndex - 1);
        randomizedQuickSort(arr, pivotIndex + 1, high);
    }
}

// Partition function for Randomized QuickSort
private static int randomizedPartition(int[] arr, int low, int high) {
    int randomPivotIndex = new Random().nextInt(high - low + 1) + low;
    swap(arr, randomPivotIndex, high); // Move random pivot to the end
    return deterministicPartition(arr, low, high); // Use the same deterministic partition
}

// Utility function to swap two elements in an array
private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

// Function to generate a random array of given size
private static int[] generateRandomArray(int size) {
    int[] arr = new int[size];
    Random random = new Random();
    for (int i = 0; i < size; i++) {
        arr[i] = random.nextInt(10); // Random integers between 0 and 9999
    }
    return arr;
}

public static void main(String[] args) {
    int size = 10; // Size of the array for analysis
    int[] arr1 = generateRandomArray(size);
    int[] arr2 = Arrays.copyOf(arr1, arr1.length); // Copy of arr1 for comparison

    // Measure time for Deterministic QuickSort
    long startTime = System.nanoTime();
    deterministicQuickSort(arr1, 0, arr1.length - 1);
    long endTime = System.nanoTime();
    System.out.println("Time taken by Deterministic QuickSort: " + (endTime - startTime) / 1e6 + "
ms");

    // Measure time for Randomized QuickSort
    startTime = System.nanoTime();
    randomizedQuickSort(arr2, 0, arr2.length - 1);
    endTime = System.nanoTime();

```

```

        System.out.println("Time taken by Randomized QuickSort: " + (endTime - startTime) / 1e6 + "
ms");
    }
}
/*
### Summary Table

| Algorithm          | Best Time Complexity | Average Time Complexity | Worst Time Complexity |
Space Complexity (Average) | Space Complexity (Worst) |
|-----|-----|-----|-----|-----|
| Deterministic QuickSort |  $\mathcal{O}(n \log n)$  |  $\mathcal{O}(n \log n)$  |  $\mathcal{O}(n^2)$  |  $\mathcal{O}(\log n)$  |
| Randomized QuickSort |  $\mathcal{O}(n \log n)$  |  $\mathcal{O}(n \log n)$  |  $\mathcal{O}(n^2)$  |  $\mathcal{O}(\log n)$  |
*/

```



Code Explanation: Deterministic and Randomized QuickSort in Java

The provided code implements both **Deterministic QuickSort** and **Randomized QuickSort**, two variations of the QuickSort algorithm. Let's go through the code line by line for a detailed explanation:

Import Statements

```

java

import java.util.Arrays;
import java.util.Random;

```

- **Arrays**: This is a utility class from Java's standard library that provides methods to manipulate arrays. In this code, it is used for creating array copies.
- **Random**: This is used to generate random numbers. It is utilized in the Randomized QuickSort to select a random pivot.

Class Definition: DAA_6_QuickSort

```

java

public class DAA_6_QuickSort {

```

This is the main class where both the deterministic and randomized versions of QuickSort are implemented.

Deterministic QuickSort

java

```
public static void deterministicQuickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivotIndex = deterministicPartition(arr, low, high);
        deterministicQuickSort(arr, low, pivotIndex - 1);
        deterministicQuickSort(arr, pivotIndex + 1, high);
    }
}
```

- **Functionality:** This is the main function for **Deterministic QuickSort**. It recursively sorts the array by partitioning it around a pivot element.
 - The base case is when `low >= high` (the array or subarray is already sorted).
 - The array is partitioned using the `deterministicPartition` function, and the subarrays (left and right of the pivot) are recursively sorted.

Deterministic Partition Function

java

```
private static int deterministicPartition(int[] arr, int low, int high) {
    int pivot = arr[high]; // using the last element as pivot
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return i + 1;
}
```

- **Functionality:** This function partitions the array based on the pivot (chosen as the last element).
 - `int pivot = arr[high]`: The pivot is chosen as the last element of the array or subarray.
 - `int i = low - 1`: `i` keeps track of the boundary between elements smaller than the pivot and those larger than the pivot.
 - **Loop** (`for`): This loop iterates through the array, and if an element is smaller than or equal to the pivot, it swaps it to the left side (so all smaller elements are grouped on the left side of `i`).
 - After the loop, the pivot is placed at the correct position by swapping it with the element at `i + 1`.
 - **Return Value:** The function returns `i + 1`, which is the index of the pivot in the partitioned array.

Randomized QuickSort

```
java
```

```
public static void randomizedQuickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivotIndex = randomizedPartition(arr, low, high);
        randomizedQuickSort(arr, low, pivotIndex - 1);
        randomizedQuickSort(arr, pivotIndex + 1, high);
    }
}
```

- **Functionality:** This is the main function for **Randomized QuickSort**, which works similarly to Deterministic QuickSort, except that the pivot is selected randomly, rather than being the last element of the subarray.
 - The base case is the same as in deterministic quicksort.
 - The array is partitioned using the `randomizedPartition` function, and subarrays are recursively sorted.

Randomized Partition Function

```
java
```

```
private static int randomizedPartition(int[] arr, int low, int high) {
    int randomPivotIndex = new Random().nextInt(high - low + 1) + low;
    swap(arr, randomPivotIndex, high); // Move random pivot to the end
    return deterministicPartition(arr, low, high); // Use the same deterministic partition
}
```

- **Functionality:** This function selects a random pivot from the array and then applies the deterministic partitioning strategy.
 - `randomPivotIndex = new Random().nextInt(high - low + 1) + low`: A random index between `low` and `high` is selected.
 - `swap(arr, randomPivotIndex, high)`: The randomly selected pivot is swapped with the last element of the subarray so that the deterministic partitioning function can be used.
 - `deterministicPartition(arr, low, high)`: The partitioning is then done using the same deterministic method as before.

Swap Function

```
java
```

```
private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

- **Functionality:** This function swaps the elements at indices `i` and `j` in the array. It is used in both partition functions to rearrange the array.

Array Generation Function

```
java

private static int[] generateRandomArray(int size) {
    int[] arr = new int[size];
    Random random = new Random();
    for (int i = 0; i < size; i++) {
        arr[i] = random.nextInt(10); // Random integers between 0 and 9999
    }
    return arr;
}
```

- **Functionality:** This utility function generates an array of random integers between 0 and 9 (you can modify the upper bound if needed).
 - It initializes an array of the given size and populates it with random integers using `random.nextInt(10)`.

Main Method

```
java

public static void main(String[] args) {
    int size = 10; // Size of the array for analysis
    int[] arr1 = generateRandomArray(size);
    int[] arr2 = Arrays.copyOf(arr1, arr1.length); // Copy of arr1 for comparison

    // Measure time for Deterministic QuickSort
    long startTime = System.nanoTime();
    deterministicQuickSort(arr1, 0, arr1.length - 1);
    long endTime = System.nanoTime();
    System.out.println("Time taken by Deterministic QuickSort: " + (endTime - startTime) /
1e6 + " ms");

    // Measure time for Randomized QuickSort
    startTime = System.nanoTime();
    randomizedQuickSort(arr2, 0, arr2.length - 1);
    endTime = System.nanoTime();
    System.out.println("Time taken by Randomized QuickSort: " + (endTime - startTime) / 1e
6 + " ms");
}
```

- **Functionality:** The `main` method runs both QuickSort algorithms and compares their performance.
 - `int size = 10`: The size of the array to be sorted is set to 10.
 - `arr1 = generateRandomArray(size)`: Generates a random array of size 10 for sorting using Deterministic QuickSort.
 - `arr2 = Arrays.copyOf(arr1, arr1.length)`: Creates a copy of the array to use it for Randomized QuickSort.
 - **Time Measurement:** The time taken to sort the array is measured using `System.nanoTime()`. The difference between start and end times is printed in milliseconds (`ms`).
 - Both sorting algorithms are applied to the respective arrays, and the time taken for each is printed.

Summary Table for Time and Space Complexity

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Space Complexity (Average)	Space Complexity (Worst)
Deterministic QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	$O(n)$
Randomized QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	$O(n)$

- **Best/Average Time Complexity:** Both deterministic and randomized QuickSort have average and best time complexity of $O(n \log n)$.
- **Worst Time Complexity:** In the worst case (if the pivot is chosen poorly), both algorithms have a time complexity of $O(n^2)$, although Randomized QuickSort generally avoids this by randomizing the pivot selection.
- **Space Complexity:** Both algorithms have $O(\log n)$ space complexity on average, but in the worst case (due to recursion), it can be $O(n)$.

Conclusion:

This code compares **Deterministic QuickSort** and **Randomized QuickSort**, both of which are efficient algorithms for sorting, but Randomized QuickSort can perform better in practice by minimizing the chances of encountering worst-case scenarios.



Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.



Here are some potential questions that may be asked during the viva based on your **QuickSort** implementation, both deterministic and randomized variants, along with appropriate answers.

1. What is QuickSort, and how does it work?

Answer: QuickSort is a **divide-and-conquer** sorting algorithm that works by selecting a **pivot** element and partitioning the array into two sub-arrays: one with elements smaller than the pivot and one with elements greater than the pivot. The algorithm recursively sorts the sub-arrays. The pivot can be selected using different strategies, such as choosing the first element, the last element, or a random element.

2. Explain the difference between deterministic and randomized QuickSort.

Answer:

- **Deterministic QuickSort:** The pivot is chosen based on a fixed rule, such as always selecting the **last element** of the array as the pivot. This can lead to poor performance ($O(n^2)$) if the array is

already sorted or nearly sorted.

- **Randomized QuickSort:** Instead of always using the last element as the pivot, a **random element** is selected as the pivot. This reduces the likelihood of encountering worst-case performance, ensuring a more balanced partitioning on average.

3. What is the time complexity of QuickSort?

Answer:

- **Best case:** $O(n \log n)$ – when the pivot divides the array into roughly equal parts.
- **Average case:** $O(n \log n)$ – expected case where the pivot divides the array reasonably well.
- **Worst case:** $O(n^2)$ – occurs when the pivot selection is poor (e.g., always picking the smallest or largest element).
- **Space complexity:** $O(\log n)$ on average for recursion stack space, but $O(n)$ in the worst case if the recursion goes too deep.

4. Why does the randomized variant of QuickSort perform better than the deterministic variant in most cases?

Answer: The randomized variant of QuickSort **minimizes the chance** of encountering the worst-case scenario ($O(n^2)$), which occurs when the pivot is poorly chosen (e.g., always the smallest or largest element). By randomly selecting the pivot, the algorithm is less likely to be affected by the order of elements in the input, making it more robust for varied data distributions.

5. Explain the partitioning step in QuickSort.

Answer: In QuickSort, **partitioning** involves rearranging the elements of the array around a pivot. The goal is to place elements smaller than the pivot to the left of the pivot and elements larger than the pivot to the right. The partitioning function returns the final position of the pivot. The algorithm then recursively sorts the left and right partitions.

6. What is the role of the pivot in QuickSort, and how is it chosen in the deterministic and randomized versions?

Answer:

- In **deterministic QuickSort**, the pivot is typically the **last element** of the array or sub-array.
- In **randomized QuickSort**, a **random element** is chosen as the pivot, and it is then swapped with the last element of the array for partitioning. The pivot's choice impacts how well the array is partitioned and, consequently, the algorithm's performance.

7. What happens if QuickSort selects a bad pivot consistently?

Answer: If QuickSort consistently selects a bad pivot (such as the smallest or largest element), the algorithm may degenerate to the worst-case time complexity of $O(n^2)$. This happens when the array is divided into highly unbalanced partitions, leading to deep recursion.

8. Explain the time complexity of the partition function.

Answer: The **partition function** takes linear time, i.e., $O(n)$. It iterates over the array to compare each element with the pivot. If an element is smaller than or equal to the pivot, it is swapped into the left side. At the end, the pivot is swapped into its correct position. Since every element is checked once, the time complexity of partitioning is linear.

9. How does recursion work in QuickSort, and what happens when the base case is reached?

Answer: QuickSort works recursively by dividing the problem into two smaller sub-problems (sorting the left and right partitions). The recursion continues until the base case is reached, where the sub-array has zero or one element, at which point it is already considered sorted, and no further recursive calls are made.

10. What is the purpose of the `swap` function in QuickSort?

Answer: The **swap** function is used to exchange the positions of two elements in the array. This is essential during partitioning, where elements smaller than the pivot are moved to the left and elements greater than the pivot are moved to the right.

11. Why do you use `Arrays.copyOf` in the `main` function when testing both variants?

Answer: The `Arrays.copyOf` method is used to create a **copy of the original array** so that both deterministic and randomized QuickSort algorithms can be tested on the **same input data**. This ensures that the comparison between the two algorithms is fair, as both algorithms work on identical initial arrays.

12. What is the difference between using `new Random().nextInt()` and `Math.random()` for generating random numbers?

Answer: Both `new Random().nextInt()` and `Math.random()` generate random numbers, but they have slight differences:

- `new Random().nextInt()` provides more control over the random number generation, such as defining the range.
- `Math.random()` generates a floating-point number between 0 and 1 and requires additional manipulation for integer ranges.

In the given code, `new Random().nextInt()` is used to generate a random index within the range of the array.

13. What is the impact of array size on the performance of QuickSort?

Answer: As the array size increases, QuickSort performs better compared to other sorting algorithms like **Bubble Sort** or **Insertion Sort**. However, if the pivot selection is poor (e.g., always picking the first or last element), performance may degrade to $O(n^2)$. Randomized QuickSort significantly reduces the chance of this happening.

14. How does the Randomized QuickSort handle the worst-case scenario better than the Deterministic QuickSort?

Answer: Randomized QuickSort mitigates the risk of worst-case time complexity by **randomly selecting the pivot**, thus reducing the chance of consistently selecting the worst pivot (e.g., the smallest or largest element). This ensures that the expected time complexity remains close to $O(n \log n)$ even for sorted or nearly sorted data.

15. What are the advantages and disadvantages of QuickSort?

Answer:

- **Advantages:**
 - QuickSort is **efficient** with an average time complexity of $O(n \log n)$.
 - It works well with large datasets.
 - It's **in-place**, requiring only a small amount of additional memory for recursion.
- **Disadvantages:**
 - The worst-case time complexity can be $O(n^2)$ with poor pivot selection.
 - It's not stable (it doesn't maintain the relative order of equal elements).

16. What is the space complexity of QuickSort?

Answer: The space complexity of QuickSort is $O(\log n)$ on average due to the recursion stack. In the worst case, if the recursion depth is large (for example, when the pivot is poorly chosen), the space complexity can be $O(n)$.

These questions should cover the key concepts, implementation details, and performance aspects of the **QuickSort** algorithm, both deterministic and randomized variants.

ChatGPT can make mistakes. Check important info.