

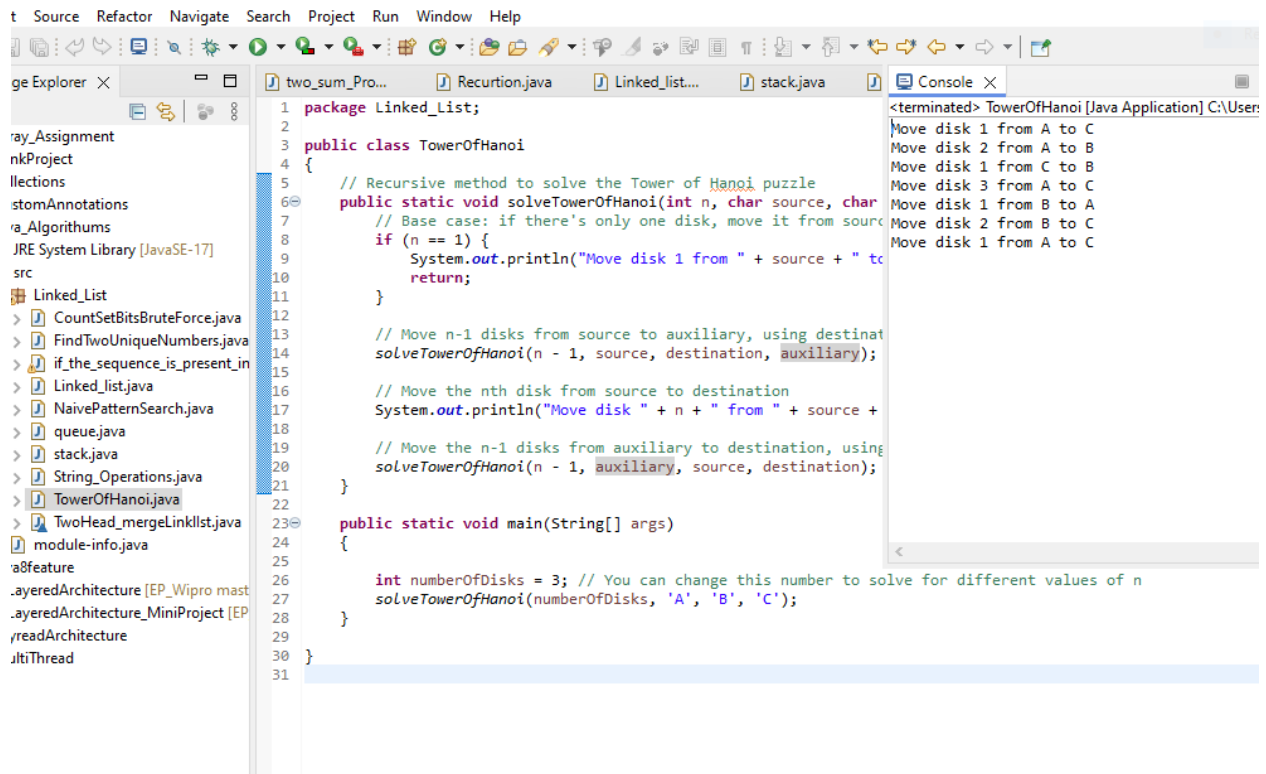
Day 13&14:

Task 1: Tower of Hanoi Solver:

Create a program that solves the Tower of Hanoi puzzle for n disks. The solution should use recursion to move disks between three pegs (source, auxiliary, and destination) according to the game's rules. The program should print out each move required to solve the puzzle.

The Tower of Hanoi puzzle involves moving a set of disks from one peg to another, with the help of an auxiliary peg, following these rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the top disk from one of the stacks and placing it on top of another stack or on an empty peg.
3. No disk may be placed on top of a smaller disk.



```
1 package Linked_List;
2
3 public class TowerOfHanoi
4 {
5     // Recursive method to solve the Tower of Hanoi puzzle
6     public static void solveTowerOfHanoi(int n, char source, char destination, char auxiliary)
7     {
8         // Base case: if there's only one disk, move it from source to destination
9         if (n == 1) {
10             System.out.println("Move disk 1 from " + source + " to " + destination);
11             return;
12         }
13
14         // Move n-1 disks from source to auxiliary, using destination as a buffer
15         solveTowerOfHanoi(n - 1, source, destination, auxiliary);
16
17         // Move the nth disk from source to destination
18         System.out.println("Move disk " + n + " from " + source + " to " + destination);
19
20         // Move the n-1 disks from auxiliary to destination, using source as a buffer
21         solveTowerOfHanoi(n - 1, auxiliary, source, destination);
22     }
23
24     public static void main(String[] args)
25     {
26         int numberOfDisks = 3; // You can change this number to solve for different values of n
27         solveTowerOfHanoi(numberOfDisks, 'A', 'B', 'C');
28     }
29 }
30
31
```

Console Output:

```
<terminated> TowerOfHanoi [Java Application] C:\User:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

❓ Base Case:

- If there is only one disk ($n == 1$), the function prints a move from the source peg to the destination peg and returns. This is the simplest scenario and terminates the recursion.

❓ Recursive Case:

- Move $n-1$ disks from the source peg to the auxiliary peg, using the destination peg as a buffer.
- Print the move of the n th (largest) disk from the source peg to the destination peg.
- Move the $n-1$ disks from the auxiliary peg to the destination peg, using the source peg as a buffer.

❓ Function Parameters:

- n : Number of disks to move.
- $source$: The peg from which disks are moved.
- $auxiliary$: The peg used as a temporary holding area.
- $destination$: The peg to which disks are moved.

Task 2: Traveling Salesman Problem

Create a function `int FindMinCost(int[] graph)` that takes a 2D array representing the graph where `graph[i][j]` is the cost to travel from city `i` to city `j`. The function should return the minimum cost to visit all cities and return to the starting city. Use dynamic programming for this solution.

- `graph`: The adjacency matrix representing the cost between cities.
- `n`: The number of cities.
- `VISITED_ALL`: A bitmask where all cities have been visited, calculated as $(1 << n) - 1$.
- `dp`: A memoization table where `dp[i][j]` represents the minimum cost to complete the tour starting from city `i` with a set of visited cities represented by the bitmask `j`.

Dynamic Programming Function `tsp`:

- **Parameters:**
 - `pos`: The current city position.
 - `mask`: The bitmask representing the set of visited cities.
- **Base Case:** If all cities have been visited (`mask == VISITED_ALL`), return the cost to go back to the starting city.
- **Memoization:** If the result for the current state (`pos, mask`) has already been computed, return the cached value.
- **Recursion:** Try to go to all unvisited cities and calculate the minimum cost recursively. Update `minCost` with the minimum value found.

Main Method:

- Creates a `TravelingSalesman` instance with the given graph.
- Calls `findMinCost` to compute and print the minimum cost to complete the tour.

Task 3: Job Sequencing Problem

Define a class `Job` with properties `int Id`, `int Deadline`, and `int Profit`. Then implement a function `List<Job> JobSequencing(List<Job> jobs)` that takes a list of jobs and returns the maximum profit sequence of jobs that can be done before the deadlines. Use the greedy method to solve this problem.

To solve the Job Sequencing problem using the greedy method, we can follow these steps:

1. **Sort Jobs by Profit:** Sort the jobs in descending order of profit.
2. **Schedule Jobs:** Iterate over the sorted jobs and schedule each job in the latest possible time slot before its deadline if that slot is available.

❓ Job Class:

- The Job class has three properties: Id, Deadline, and Profit.
- It implements the Comparable<Job> interface to allow sorting by profit in descending order.

❓ Job Sequencing Function:

- **Sorting:** The jobs are sorted by profit in descending order using Collections.sort(jobs).
- **Maximum Deadline:** Find the maximum deadline among all jobs to determine the size of the scheduling array.
- **Result Array and Slot Array:**
 - result array to store the scheduled jobs.
 - slot array to track which time slots are occupied.
- **Scheduling Jobs:** Iterate through the sorted jobs and try to schedule each job in the latest possible time slot before its deadline.
 - If a free slot is found, mark the slot as occupied and store the job in the result array.
- **Collect Scheduled Jobs:** After scheduling, collect the jobs from the result array into a list to return.

❓

