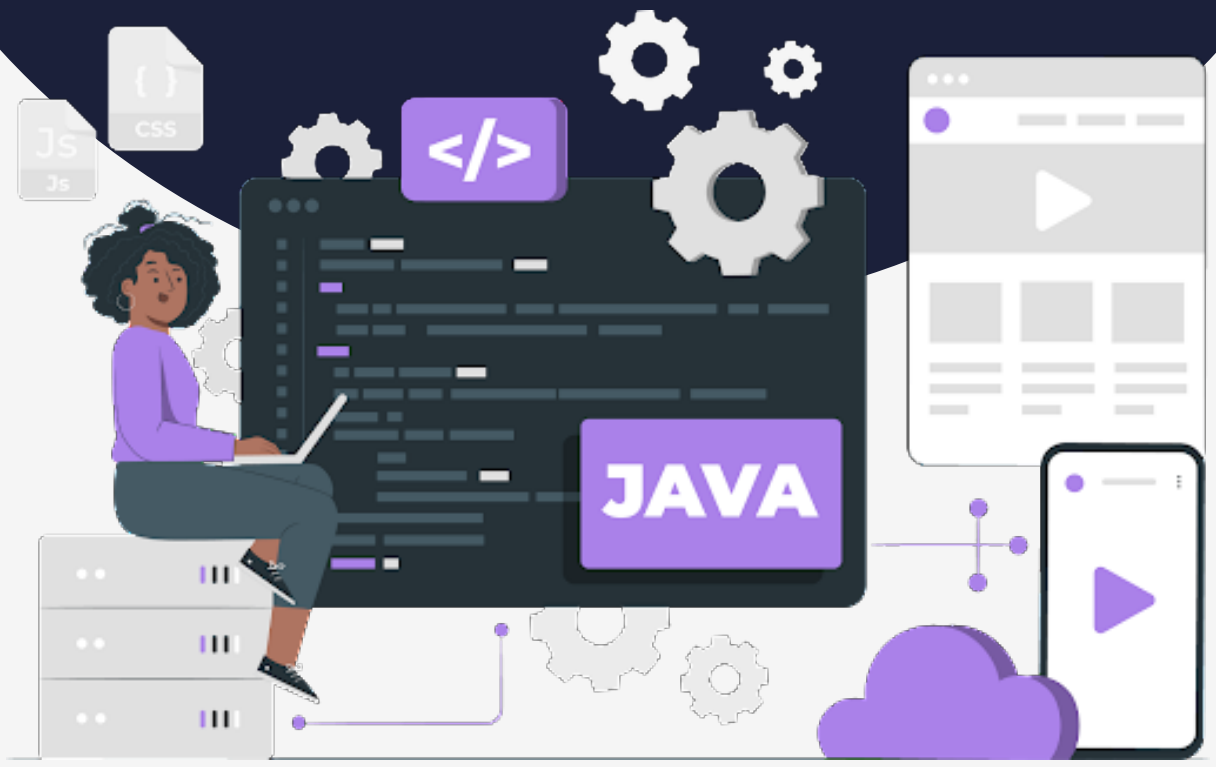
Lesson:

Greedy Algorithms



Pre Requisites:

- Sorting algorithms
- Basic JAVA syntax

List of concepts involved :

- Introduction to greedy algorithm
- Job scheduling problem
- Merge interval problem
- Fractional knapsack problem

What is a greedy algorithm?

The greedy method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results.

The Greedy method is the simplest and straightforward approach.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Job scheduling problem :

Q1. Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. Maximize the total profit if only one job can be scheduled at a time.

Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Output: Following is maximum profit sequence of jobs: c, a

Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
1	2	100
2	1	19
3	2	27
4	1	25
5	3	15

Output: Following is maximum profit sequence of jobs: 3, 1, 5

Solution :

Code : LP_Code1.java

Output :

```
The maximum observed profit is :
d a b
```

Approach :
Here we will take a greedy approach to implement the job scheduling problem. We will follow the following steps to schedule the job in the desired ways.

- First, sort the jobs in the decreasing order of their profits.
- Then find the highest deadline among all deadlines.
- Next, we need to assign time slots to individual job ids.
- First, check if the maximum possible time slot for the job, i.e., its deadline, is assigned to another job or not. If it is not filled yet, assign the slot to the current job id.
- Otherwise, search for any empty time slot less than the deadline of the current job. If such a slot is found, assign it to the current job id and move to the next job id.

Merge Interval problem :

Q2. Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.
Example 1:
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

Example 2:
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.

Solution :

Code : LP_Code2.java

Output :

```
The Merged Intervals are: [1,9][12,14]
```

Approach :

- Sort the intervals based on the beginning of each interval.
- There is overlap when beginning of current interval<= ending of previous interval
- In that case, the ending of the merged interval is maximum of both the ending. Beginning of the merged interval was the same as the beginning of the previous interval.
- Discussing the 3 possibilities :
  - If the current interval's end is not greater than the last interval's end, that means the current interval is inside/overlapping with the last interval, so we do not need to do anything.

- If there is an intersection between current and last interval then we can just update the last interval's end.
- If current interval is not inside the last interval, and has no intersection with last interval, then we add it into the result list

Fractional knapsack problem :

Q3. Given the weights and values of N items, in the form of {value, weight} put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.
Input: arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50
Output: 240
Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.
Hence total price will be 60+100+(2/3)(120) = 240

Input: arr[] = {{500, 30}}, W = 10
Output: 166.667

solution :

Code : LP_code3.java

Output :

```
118.0
```

Approach :

- For each item, compute its value / weight ratio.
- Arrange all the items in decreasing order of their value / weight ratio.
- Start putting the items into the knapsack beginning from the item with the highest ratio.
- Put as many items as you can into the knapsack.

Next class teaser:

- Tree