Marwadi University Faculty of Engineering and Technology Department of Information and Communication Technology		
Subject: DAA (01CT0512)	**AIM: Implementing application-based algorithms using Greedy Approach**	
Experiment No: 07	**Date:**	**Enrolment No: 92301733046**

1. Job Scheduling Problem

Theory: This method solves the **Job Scheduling Problem with Deadlines and Profits** using a greedy approach. Jobs are first sorted by profit in descending order, and then each job is scheduled in the latest available slot before its deadline. This ensures maximum profit while meeting job deadlines.

- Greedy Strategy
- Maximizing profits under deadline

Programming Language: Python **Code:**

```
def jobScheduling(jobs, deadlines, profits):
   n = len(jobs)
   job_data = list(zip(jobs, deadlines, profits))
   job_data.sort(key=lambda x: x[2], reverse=True) max_deadline
   = max(deadlines)
   schedule = [0] * max_deadline job_sequence =
   [None] * max_deadline total_profit = 0
   for job, deadline, profit in job_data:
      for j in range(min(deadline, max_deadline) - 1, -1, -1): if
         schedule[j] == 0:
            schedule[j] = 1 job_sequence[j]
            = job total_profit += profit
            break
   print("Scheduled jobs:", [job for job in job_sequence if job is not None]) print("Max
   profit:", total_profit)
   return job_sequence
jobs = ["a", "b", "c", "d", "e","f"]
deadlines = [2, 1, 3, 1, 2,1]
profits = [100, 20, 35, 27, 30,28]
jobScheduling(jobs, deadlines, profits)
```

Output:

```
Scheduled jobs: ['e', 'a', 'c']
Max profit: 165
```

Space complexity: O(n + d)

Justification: The job_data list stores all jobs with deadlines and profits. The schedule and job_sequence arrays have size equal to the maximum deadline d.

Time complexity: O(nlogn + n.d) => O(n^2)

Best case time complexity: O(nlogn)

Justification: When deadlines are small or all jobs can be scheduled easily, sorting dominates the work

Worst case time complexity: O(nlogn + n.d) => O(n^2)

Justification: If every job has a large deadline, we still sort first (O(n log n)) and then for each job may scan up to d slots, giving O(n·d).

2. Activity Selection

Theory: This method solves the **Activity Selection Problem** using a greedy approach. Activities are sorted by their finishing times, and then the earliest finishing activity is selected first. After that, each next activity is chosen only if its start time is greater than or equal to the finish time of the previously selected activity. This ensures the maximum number of non-overlapping activities are selected.

- Greedy Strategy
- Maximizing number of compatible activities

Programming Language: Python

Code:

```
def activity_selection(start, end):
    n = len(start)
    activities = list(zip(end, start)) # (end, start) activities.sort()  # sort
    by end time
    selected = [] last_end
    = -1
                for e, s in activities:
                  if s >= last_end:
          selected.append((s, e)) last_end = e
    print("Selected activities (start, end):", selected) start = [5, 6,
0, 4, 10]
end = [8, 10, 5, 7, 12]
activity_selection(start, end)
```

Output:

```
PS D:\DAA\Lab and Lecture Codes> python -u "d:\DAA\Lab and Lecture Codes\Activity_selection.py"
Selected activities (start, end): [(0, 5), (5, 8), (10, 12)]
PS D:\DAA\Lab and Lecture Codes>
```

Space complexity: O(n)

Justification: The algorithm stores activities in a list of pairs (end, start). The selected activities are stored in another list. Both grow linearly with the number of activities, so the space complexity is O**(n)**.

Time complexity: O(nlogn)

Best case time complexity: O(nlogn)

Justification: Regardless of how start and end times are distributed, sorting is always needed, and then we check each activity once.

Worst case time complexity: O(nlogn)

Justification: Even if all activities overlap heavily or all are non-overlapping, we still sort all activities and then process each one.