Experiment #6		Student ID	
Date		Student Name	

Experiment Title: Implementation of Programs on Greedy method Problems - Job Sequence with Deadlines and Knapsack Problems.

Aim/Objective: Students can able to apply and analyze the Job Sequence with Deadlines and knapsack problems on greedy method. The aim of these algorithm to find the optimal solution.

Description: The Job Sequencing with Deadlines problem is a scheduling problem where the goal is to maximize the total profit by selecting a subset of jobs to complete within their respective deadlines. Each job has a deadline and a profit associated with it.

The Knapsack Problem is a classic optimization problem where the goal is to maximize the total profit of items that can be placed into a knapsack of fixed weights. Each item has a specific weight and profit, and the knapsack has a weight limit.

Pre-Requisites:

Pre-Lab:

. Given the jobs, their deadlines and associated profits as shown:

Jobs, J1, J2, J3, J4, J5, J6

Deadlines, 5, 3, 3, 2, 4, 2

Profits, 200, 180, 190, 300, 120, 100

Answer the following questions:

Write the optimal schedule that gives maximum profit.

Are all the jobs completed in the optimal schedule?

What is the maximum earned profit?

- **Procedure/Program:**

optimal schedule:

2 2 2 2

- Job sequence: J4, J3, J2, J5, J1
- This sequence maximizes Profit based on deadlines and job availability.

Job completion:

- Not completed: J6 is not completed as it's excluded in the optimal schedule

Maximum Earned Profit:

- Total Profit: 990 units.

Experiment #6		Student ID	
Date		Student Name	

- **Data and Results:**

Data: Jobs with deadlines and profits are given to maximize earnings

Result: optimal Job sequence provides maximum profit of 990 units

- **Analysis and Inferences:**

Analysis: greedy algorithm prioritizes high-profit jobs within given deadlines efficiently

Inferences: Not all jobs are completed, but maximum profit is achieved.

2. Explain why 0-1 Knapsack problems cannot be solved using greedy method unlike fractional knapsack. (Students may attempt this exercise after completion of 0/1 knapsack in dynamic approach)

- **Procedure/Program:**

The 0-1 knapsack problem requires whole item selection, unlike fractional knapsack. greedy choices may overlook better combinations, needing dynamic programming for optimal solutions.

- **Data and Results:**

Data: 0-1 knapsack involves whole item selection, unlike fractional knapsack.

Result:
greedy method fails for 0-1 knapsack, dynamic Programming needed

- **Analysis and Inferences:**

Analysis: greedy choices overlook optimal combinations in 0-1 knapsack problems

Inferences: Dynamic Programming is essential for solving 0-1 knapsack optimally.

In-Lab:

1. Given an array of size n that has the following specifications:

   a. Each element in the array contains either a police officer or a thief.

   b. Each police officer can catch only one thief.

   c. A police officer cannot catch a thief who is more than K units away from the police officer.

   We need to find the maximum number of thieves that can be caught.

   Input

   arr [] = {'P', 'T', 'T', 'P', 'T'},

   k = 1.

   Output

   2

   Here maximum 2 thieves can be caught; first police officer catches first thief and second police officer can catch either second or third thief.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  char arr[] = {'P', 'T', 'T', 'P', 'T'};
  int n = sizeof(arr) / sizeof(arr[0]);
  int k = 1;
```

```
int Policecount = 0,   thievescount = 0,   count = 0;
 int Police[n],   thieves[n];

for(int i = 0;   i < n;   i++){
if(arr[i] == 'P')  Police[Policecount++] = i;
 else if(arr[i] == 'T')  thieves[thievescount++]=i;
 }
for(int  i=0;  j=0;  i < Policecount && j < thieves
                                        count;){

if(abs(Police[i] - thieves[j]) <= k){
      count++;
     i++; j++;
  } else if(Police[i] < thieves[j]){
       i++;
     } else   {
       j++;
     }
  }
Printf("%d\n", count);
     return 0;
  }
```

- **Data and Results:**

Data: Police officers catch thieves within a limited distance constraint efficiently

Result: Maximum number of thieves caught is determined using distance constraint

- **Analysis and Inferences:**

Analysis: Matching police officers and thieves within distance maximizes catches efficiently

inferences: greedy approcach ensures optimal thief catching within specified distance constraints.

Post-Lab:

1. 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. How to maximize total profit if only one job can be scheduled at a time.

Input

4

Job	**ID**	**Deadline Profit**
a	4	20
b	1	10
c	1	40
d	1	30

Output

60

Profit sequence of jobs is c, a

Data:
2
Jobs IDs deadlines and profits for maximum scheduling efficiency

Result:
2
Maximum Profit obtained by scheduling jobs is sixty units total

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
   struct Job {
      char id;
      int deadline;
      int Profit;
};
 int main() {
 struct Job jobs[] = {{'a', 4, 20}, {'b', 1, 10},
{'c', 1, 40}, {'d', 1, 30}};
 int n = sizeof(jobs) / sizeof(jobs[0]);
  for(int i = 0; i< n - 1; i++) {
    for(int j=0; j <n - i - 1 ; j++) {
if(jobs[j].Profit  (jobs[j+1].Profit ){
          struct job temP = jobs[j];
          jobs[j] = jobs[j+1];
          jobs[j+1] = temP;
```

- **Data and Results:**

```
    }
  }
}
```

```
int maxDeadline = 0;
for(int i = 0; i < n; i++){
if(jobs[i].deadline > maxDeadline){
maxDeadline = jobs[i].deadline;
}
}
int timeslot[maxDeadline];
memset(timeslot, -1, sizeof(timeslot));
int totalProfit = 0;
char resultJobs[n];
int resultCount = 0;
for(int i = 0; i < n; i++){
for(int j = jobs[i].deadline-1; j >= 0; j--){
if(timeslot[j] == -1){
timeslot[j] = jobs[i].id;
totalProfit += jobs[i].Profit;
resultJobs[resultCount++] = jobs[i].id;
break; } } }
Printf("output: %d\n", totalProfit);
Printf("Profit Seqvence of jobs is: ");
for(int i = 0; i < resultCount; i++){
Printf("%c", resultJobs[i]);
}
Printf("\n"); return 0; }
```

Experiment #6		Student ID	
Date		Student Name	

- **Analysis and Inferences:**

Analysis: (2) greedy scheduling (2) maximizes Profit by Prioritizing high-value jobs efficiently.

inferences: optimal jobs selection improves profit within given deadline constraints effectively.

- **Sample VIVA-VOCE Questions:**

1. Describe the steps involved in the greedy algorithm for Job Sequencing with Deadlines.
2. Why do we sort the jobs in decreasing order of profit?
3. Under what circumstances might the greedy algorithm be less effective?
4. What is the time complexity of the greedy algorithm for the Fractional Knapsack problem?
5. What is the significance of the value-to-weight ratio in the selection process?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) Sort jobs, schedule based on deadlines, maximize Profit by selecting highest-value jobs first.

2) Sorting ensures highest Profits are Prioritized for scheduling, maximizing overall Profit

3) greedy may fail when optimal solutions require combining items instead of choosing individually

4) Time complexity is O(nlogn) due to Sorting jobs by Profit

5) value-to-weight ratio guides optimal selections, maximizing Profit within weight constraints efficiently.