Batch:T6

Practical No.6

Title of Assignment: Greedy approach

Student Name: Parshwa Herwade

Student PRN: 22510064

To apply Greedy method to solve problems of

1) Job sequencing with deadlines

1.A) Generate table of feasible,proceesing sequencing , profit .

ANS.

Given the sequence of jobs with profits (p1, p2, ..., p7) = (3, 5, 20, 18, 1, 6, 30) and deadlines (d1, d2, ..., d7) = (1, 3, 4, 3, 2, 1, 2), we would arrange jobs in order of profit, attempt to schedule them within their deadline, and maximize profit.

1.B) What is the solution generated by the function JS when n=7, (p1,p2,...,p7) = (3,5,20,18,1,6,30), and (d1,d2,d3,...,d7) = (1,3,4,3,2,1,2)?

ANS.

Here is the solution breakdown for JS(n=7):

* Jobs: (3, 5, 20, 18, 1, 6, 30)
* Deadlines: (1, 3, 4, 3, 2, 1, 2)

Step-by-step greedy job selection:

* Sort jobs by profit: (p7, p3, p4, p6, p2, p1, p5) = (30, 20, 18, 6, 5, 3, 1)
* Try to schedule each job by its deadline and fit it into the sequence.

**Optimal sequence**: Job 7 → Job 3 → Job 4 → Job 2 → Job 1  
**Maximum Profit**: 79.

1.C) Input: Five Jobs with following deadlines and profits.

ANS.

JobID Deadline Profit

a 2 90

b 1 19

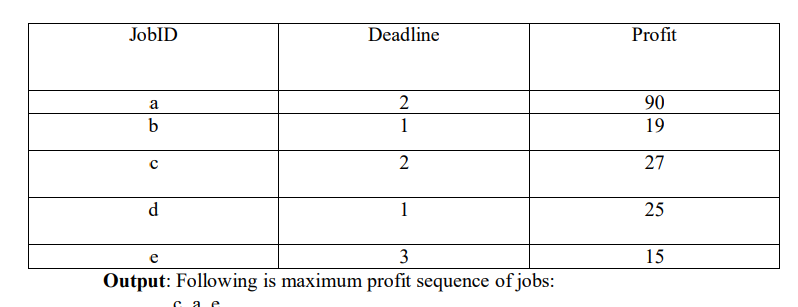
c 2 27

d 1 25

e 3 15

Output: Maximum profit sequence of jobs: c, a, e

1.D) Study and implement Disjoint set algorithm to reduce time complexity of JS from O(𝑛 2 ) to nearly O(n).



ANS. The Disjoint Set algorithm is used to optimize the job sequencing problem from O(n²) to O(n log n). Implementing it with union-find reduces unnecessary checks.

**Pseudocode:**

1. **Sort jobs by profit** in descending order.
2. **Initialize result array** to store job sequences and a boolean array slots to keep track of available time slots.
3. **Iterate over each job**:
   * For each job, try to schedule it at the latest possible time slot before its deadline.
   * If the time slot is available, assign the job to that slot.
4. **Print the job sequence** that gives the maximum profit.

function jobSequencing(jobs, n):

sort jobs by profit in descending order

result = array of size n initialized to -1

slots = array of size n initialized to false

totalProfit = 0

for each job in jobs:

for j = min(n, job.deadline) - 1 down to 0:

if slots[j] is false:

result[j] = job.id

slots[j] = true

totalProfit += job.profit

break

print "Job sequence:", result

print "Maximum Profit:", totalProfit

Code:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Job {

    char id;

    int profit;

    int deadline;

};

bool compare(Job a, Job b) {

    return (a.profit > b.profit);

}

void jobSequencing(vector<Job>& jobs, int n) {

    sort(jobs.begin(), jobs.end(), compare);

    vector<int> result(n, -1);

    vector<bool> slot(n, false);

    int totalProfit = 0;

    for (int i = 0; i < jobs.size(); i++) {

        for (int j = min(n, jobs[i].deadline) - 1; j >= 0; j--) {

            if (!slot[j]) {

                result[j] = i;

                slot[j] = true;

                totalProfit += jobs[i].profit;

                break;

            }

        }

    }

    cout << "Job sequence with maximum profit:\n";

    for (int i = 0; i < n; i++) {

        if (result[i] != -1)

            cout << jobs[result[i]].id << " ";

    }

    cout << "\nMaximum Profit: " << totalProfit << endl;

}

int main() {

    int n;

    cout << "Enter the number of jobs: ";

    cin >> n;

    vector<Job> jobs(n);

    for (int i = 0; i < n; i++) {

        cout << "Enter job ID, profit, and deadline for job " << i + 1 << ": ";

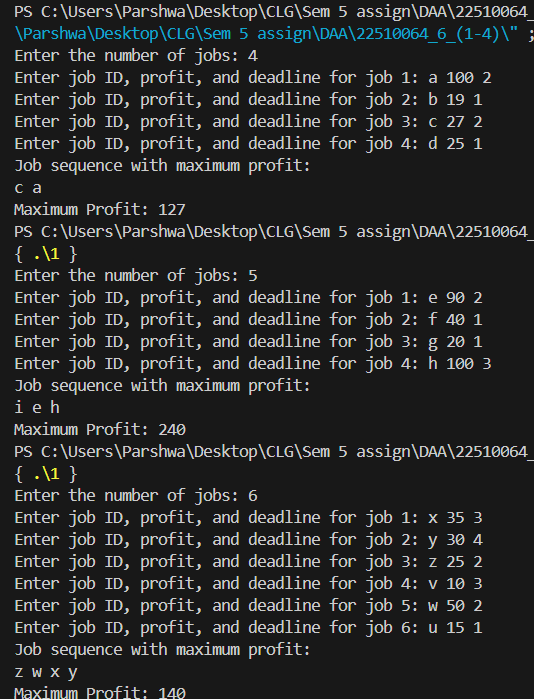
        cin >> jobs[i].id >> jobs[i].profit >> jobs[i].deadline;

    }

    jobSequencing(jobs, n);

    return 0;

}

Output:  


**Complexity:**

* **Best/Average/Worst Case**: O(n log n) (due to sorting)
* **Space Complexity**: O(n)

2) To implement Fractional Knapsack problem 3 objects (n=3). (w1,w2,w3) = (19,15,10) (p1,p2,p3) = (25,24,15) M=20 With strategy a) Largest-profit strategy b) Smallest-weight strategy c) Largest profit-weight ratio strategy.

ANS.

**Pseudocode for Largest-Profit Strategy:**

1. **Sort items** by profit in descending order.
2. **Initialize totalProfit** to zero.
3. **Iterate over each item**:
   * If the item’s weight is less than or equal to the remaining capacity, take the full item.
   * Otherwise, take the fractional part of the item that fits in the remaining capacity.
4. **Return the total profit**.

function fractionalKnapsack(capacity, items):

sort items by profit in descending order

totalProfit = 0

for each item in items:

if item.weight <= capacity:

capacity -= item.weight

totalProfit += item.profit

else:

totalProfit += (item.profit \* (capacity / item.weight))

break

return totalProfit

**Pseudocode for Smallest-Weight Strategy:**

1. **Sort items** by weight in ascending order.
2. **Repeat the same steps** as in the largest-profit strategy, but based on the sorted weights.

function fractionalKnapsack(capacity, items):

sort items by weight in ascending order

totalProfit = 0

for each item in items:

if item.weight <= capacity:

capacity -= item.weight

totalProfit += item.profit

else:

totalProfit += (item.profit \* (capacity / item.weight))

break

return totalProfit

**Pseudocode for Largest Profit-Weight Ratio Strategy:**

1. **Sort items** by profit-to-weight ratio in descending order.
2. **Repeat the same steps** as in the largest-profit strategy, but based on the sorted ratios.

function fractionalKnapsack(capacity, items):

sort items by profit-to-weight ratio in descending order

totalProfit = 0

for each item in items:

if item.weight <= capacity:

capacity -= item.weight

totalProfit += item.profit

else:

totalProfit += (item.profit \* (capacity / item.weight))

break

return totalProfit

Code:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Item {

    int weight, profit;

};

bool compareByProfit(Item a, Item b) {

    return a.profit > b.profit;

}

bool compareByWeight(Item a, Item b) {

    return a.weight < b.weight;

}

bool compareByRatio(Item a, Item b) {

    double r1 = (double)a.profit / a.weight;

    double r2 = (double)b.profit / b.weight;

    return r1 > r2;

}

double fractionalKnapsack(int capacity, vector<Item>& items, bool (\*compare)(Item, Item)) {

    sort(items.begin(), items.end(), compare);

    double totalProfit = 0.0;

    for (int i = 0; i < items.size(); i++) {

        if (items[i].weight <= capacity) {

            capacity -= items[i].weight;

            totalProfit += items[i].profit;

        } else {

            totalProfit += (double)items[i].profit \* ((double)capacity / items[i].weight);

            break;

        }

    }

    return totalProfit;

}

int main() {

    int n, capacity;

    cout << "Enter the number of items: ";

    cin >> n;

    vector<Item> items(n);

    for (int i = 0; i < n; i++) {

        cout << "Enter weight and profit for item " << i + 1 << ": ";

        cin >> items[i].weight >> items[i].profit;

    }

    cout << "Enter the capacity of the knapsack: ";

    cin >> capacity;

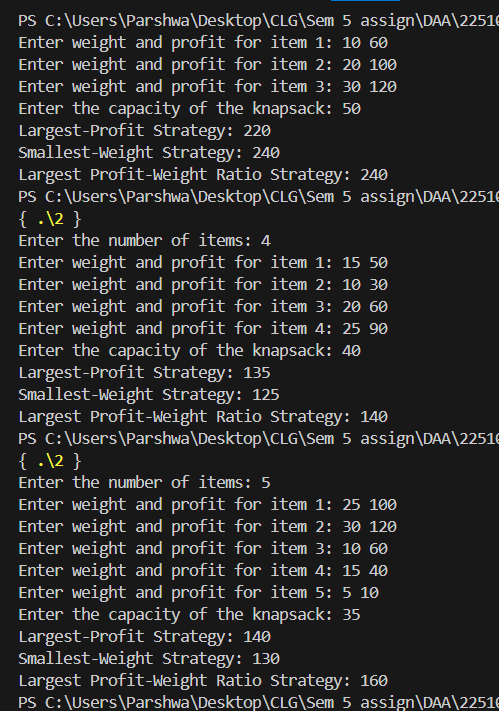
    cout << "Largest-Profit Strategy: " << fractionalKnapsack(capacity, items, compareByProfit) << endl;

    cout << "Smallest-Weight Strategy: " << fractionalKnapsack(capacity, items, compareByWeight) << endl;

    cout << "Largest Profit-Weight Ratio Strategy: " << fractionalKnapsack(capacity, items, compareByRatio) << endl;

    return 0;

}

Output:  
 **Complexity:**

* **Best/Average/Worst Case Time Complexity**: O(n log n) (due to sorting)
* **Space Complexity**: O(n)