**Batch : T7**

**Practical No. 6**

**Title of Assignment : Greedy Method**

**Student Name: Om Kulkarni**

**Student PRN: 22510034**

**Problem Statement:**

To apply Greedy method to solve problems of

1) Job sequencing with deadlines

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

Sequence No Sequence Profit Calculation Total Profit

1 c, a, e 27 (c) + 90 (a) + 15 (e) 132

2 a, c, e 90 (a) + 27 (c) + 15 (e) 132

3 a, d, e 90 (a) + 25 (d) + 15 (e) 130

4 c, b, e 27 (c) + 19 (b) + 15 (e) 61

5 d, c, e 25 (d) + 27 (c) + 15 (e) 67

6 b, c, e 19 (b) + 27 (c) + 15 (e) 61

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)?

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

|  |  |  |
| --- | --- | --- |
| JobID | Deadline | Profit |
| a | 2 | 90 |
| b | 1 | 19 |
| c | 2 | 27 |
| d | 1 | 25 |
| e | 3 | 15 |

**Output**: Following is maximum profit sequence of jobs:

c, a, e

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

1. Algorithm/Pseudocode

**Input**:

* Read the number of jobs n and for each job i, read the job's id, deadline, and profit.

**Sort Jobs by Profit**:

* Sort the jobs in descending order of profit. This ensures that we maximize profit by scheduling higher-profit jobs first.

**Initialize Disjoint Set Union (DSU)**:

* Find the maximum deadline among all jobs: maxDeadline = max(job.deadline for each job).
* Create a DSU array parent[] with size maxDeadline + 1. Initialize each parent array element such that parent[i] = i, which means each slot is available.

**Job Selection**:

* For each job in the sorted list:
  1. Use the **find** operation to find the largest available slot x that is less than or equal to the job's deadline.
  2. If a valid slot x is found (i.e., x > 0), schedule the job in that slot.
  3. Use the **union** operation to mark the slot x as occupied, linking it to slot x-1.
  4. Add the job's profit to the total profit.
  5. Add the job to the sequence of scheduled jobs.

**Output**:

* Print the job sequence.
* Print the total profit.

Program Code

#include <iostream>

#include <algorithm>

using namespace std;

// Function to find the maximum profit sequence of jobs

void jobSequencing(char *jobID*[], int *deadlines*[], int *profits*[], int *n*) {

    // Arrays to keep track of used time slots and job sequence

    bool slot[*n*] = {false};  // Keep track of filled time slots

    char jobSequence[*n*];     // Store job sequence

    // Initialize total profit

    int totalProfit = 0;

    // Sort jobs based on profits in descending order using a simple bubble sort

    for (int i = 0; i < *n* - 1; i++) {

        for (int j = 0; j < *n* - i - 1; j++) {

            if (*profits*[j] < *profits*[j + 1]) {

                swap(*profits*[j], *profits*[j + 1]);

                swap(*deadlines*[j], *deadlines*[j + 1]);

                swap(*jobID*[j], *jobID*[j + 1]);

            }

        }

    }

    // Iterate over all jobs

    for (int i = 0; i < *n*; i++) {

        // Find the latest available time slot for the job

        for (int j = min(*n*, *deadlines*[i]) - 1; j >= 0; j--) {

            if (!slot[j]) {  // If slot is free

                slot[j] = true;       // Mark the slot as filled

                jobSequence[j] = *jobID*[i];  // Assign the job to this slot

                totalProfit += *profits*[i];  // Add profit to total

                break;

            }

        }

    }

    // Display the job sequence and total profit

    cout << "Job Sequence: ";

    for (int i = 0; i < *n*; i++) {

        if (slot[i]) {

            cout << jobSequence[i] << " ";

        }

    }

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

}

int main() {

    // Example 1: 7 Jobs with deadlines and profits

    char jobID1[] = {'1', '2', '3', '4', '5', '6', '7'};

    int deadlines1[] = {1, 3, 4, 3, 2, 1, 2};

    int profits1[] = {3, 5, 20, 18, 1, 6, 30};

    int n1 = sizeof(jobID1) / sizeof(jobID1[0]);

    cout << "Test Case 1:\n";

    jobSequencing(jobID1, deadlines1, profits1, n1);

    // Example 2: 5 Jobs with deadlines and profits

    char jobID2[] = {'a', 'b', 'c', 'd', 'e'};

    int deadlines2[] = {2, 1, 2, 1, 3};

    int profits2[] = {90, 19, 27, 25, 15};

    int n2 = sizeof(jobID2) / sizeof(jobID2[0]);

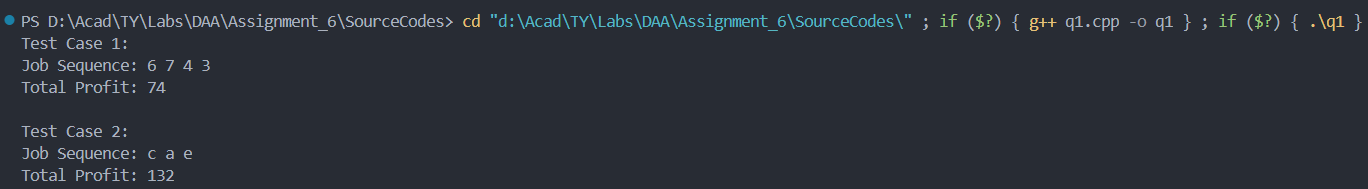
    cout << "\nTest Case 2:\n";

    jobSequencing(jobID2, deadlines2, profits2, n2);

    return 0;

}

1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

**Time Complexity:**

* Sorting jobs: O(n^2) (Bubble sort).
* Allocating jobs to slots: O(n^2).

**Space Complexity:**

* O(n) for arrays (slot[], jobSequence[]).

**Problem Statement:**

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

1. Algorithm/Pseudocode

**1. Input:**

* Three items, each with weight and profit.
* The knapsack has a maximum capacity MMM.

**2. Algorithm for Each Strategy:**

**a) Largest-Profit Strategy:**

1. **Sort Items by Profit**:
   * Sort the items in decreasing order of their profits.
2. **Select Items for the Knapsack**:
   * Add the item with the highest profit to the knapsack as long as it fits.
   * If the current item doesn't fit fully, add a fraction of it.
3. **Return the Total Profit**:
   * Return the total profit obtained by the selected items.

**b) Smallest-Weight Strategy:**

1. **Sort Items by Weight**:
   * Sort the items in increasing order of their weights.
2. **Select Items for the Knapsack**:
   * Add the item with the smallest weight to the knapsack as long as it fits.
   * If the current item doesn't fit fully, add a fraction of it.
3. **Return the Total Profit**:
   * Return the total profit obtained by the selected items.

**c) Largest Profit-Weight Ratio Strategy:**

1. **Sort Items by Profit-to-Weight Ratio**:
   * Calculate the profit-to-weight ratio for each item: ratioi=pi/wi
   * Sort the items in decreasing order of their profit-to-weight ratio.
2. **Select Items for the Knapsack**:
   * Add the item with the highest profit-to-weight ratio to the knapsack as long as it fits.
   * If the current item doesn't fit fully, add a fraction of it.
3. **Return the Total Profit**:
   * Return the total profit obtained by the selected items.

Program Code

#include <iostream>

#include <algorithm>

using namespace std;

// Function to solve knapsack with a given order of weights and profits

double fractionalKnapsack(int *w*[], int *p*[], int *n*, int *capacity*) {

    double totalValue = 0.0;  // Total value in knapsack

    int currentWeight = 0;    // Current weight in knapsack

    for (int i = 0; i < *n*; i++) {

        if (currentWeight + *w*[i] <= *capacity*) {

            // Take the whole item

            currentWeight += *w*[i];

            totalValue += *p*[i];

        } else {

            // Take a fraction of the item

            int remainingWeight = *capacity* - currentWeight;

            totalValue += (*p*[i] / (double)*w*[i]) \* remainingWeight;

            break;

        }

    }

    return totalValue;

}

int main() {

    // Input data

    int w[] = {19, 15, 10};  // Weights

    int p[] = {25, 24, 15};  // Profits

    int n = 3;               // Number of items

    int capacity = 20;       // Knapsack capacity

    // a) Largest-profit strategy

    cout << "Largest-profit strategy:\n";

    // Sort items by decreasing profit

    int indices[] = {0, 1, 2};

    sort(indices, indices + n, [&](int *i*, int *j*) { return p[*i*] > p[*j*]; });

    int wp[3], pp[3];  // Weights and profits sorted based on strategy

    for (int i = 0; i < n; i++) {

        wp[i] = w[indices[i]];

        pp[i] = p[indices[i]];

    }

    cout << "Maximum value: " << fractionalKnapsack(wp, pp, n, capacity) << "\n\n";

    // b) Smallest-weight strategy

    cout << "Smallest-weight strategy:\n";

    // Sort items by increasing weight

    sort(indices, indices + n, [&](int *i*, int *j*) { return w[*i*] < w[*j*]; });

    for (int i = 0; i < n; i++) {

        wp[i] = w[indices[i]];

        pp[i] = p[indices[i]];

    }

    cout << "Maximum value: " << fractionalKnapsack(wp, pp, n, capacity) << "\n\n";

    // c) Largest profit-weight ratio strategy

    cout << "Largest profit-weight ratio strategy:\n";

    // Sort items by decreasing profit-to-weight ratio

    sort(indices, indices + n, [&](int *i*, int *j*) { return (p[*i*] / (double)w[*i*]) > (p[*j*] / (double)w[*j*]); });

    for (int i = 0; i < n; i++) {

        wp[i] = w[indices[i]];

        pp[i] = p[indices[i]];

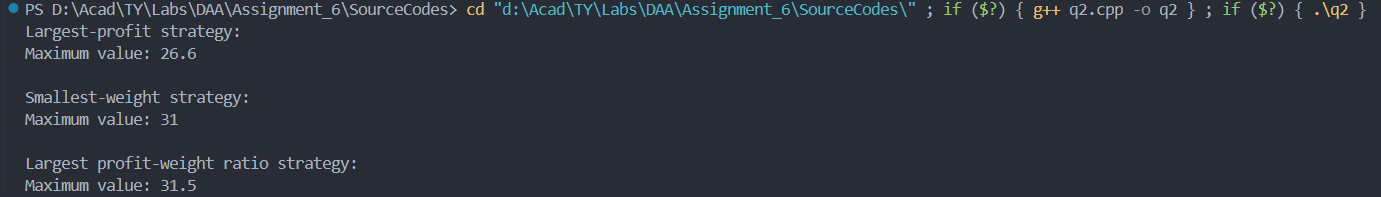
    }

    cout << "Maximum value: " << fractionalKnapsack(wp, pp, n, capacity) << "\n";

    return 0;

}

1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

**Time Complexity**:

* Sorting items: O(nlogn) where nnn is the number of items.
* Iterating through sorted items: O(n).
* Overall: O(nlogn).

**Space Complexity**:

* O(n) for storing the weights, profits, and sorted items.