CSE 208 : ALGORITHM LAB ASSIGNMENT

Assignment : 1

Submitted BY :

Name : Afrin Sultana Akhi

Reg : 22101095

Section : B2

Submitted To :

Name : Md. Shahidul Islam

1 : Algorithm Implementations :

Dynamic Programming :

```
#include <bits/stdc++.h>
using namespace std;
// Function to solve 0/1 knapsack using dynamic programming
float knapsackDP(int capacity, int items[][3], int n) {
    float dp[n + 1][capacity + 1]; //creates new array named dp
array to store the maximum profit

    // Initialize dp array
    for (int i = 0; i <= n; ++i) {
        for (int w = 0; w <= capacity; ++w) {
            if (i == 0 || w == 0) // initializes the base case of
the DP ALGO, where either there are no items or the knapsack
capacity zero, resulting in a maximum value of zero.
                dp[i][w] = 0;
            else if (items[i - 1][2] <= w)
/*checks if the weight of the current item is less than or equal
to the current capacity of the knapsack.If true, it calculates the
maximum value by either excluding the current item or including
it.*/
 dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - items[i - 1][2]] +
items[i - 1][1]);
 /*It compares the value obtained by excluding the current item
with the value obtained by including the current item and selects
the maximum of the two.*\
            else
                dp[i][w] = dp[i - 1][w];
        } }
    // Backtrack to find the selected items
    int w = capacity; //W =  remaining capacity
    cout << "Selected items (ID): ";
    for (int i = n; i > 0 && w > 0; --i)
// loop continues until either all items have been considered or
the remaining capacity becomes zero.
    {
```

```
        if (dp[i][w] != dp[i - 1][w]) {
      //current Value != prev value
//If the value of the current item is not equal to the value of
the previous item,it means that the current item is included in
the knapsack.
            cout << items[i - 1][0] << " ";
            w -= items[i - 1][2];
        }
    }
    cout << endl;

    return dp[n][capacity];//returns the maximum value that can be
obtained by selecting items .
}

int main() {
    int items[][3] = {
//here item is a 2d array where each row represents an item with
  //   ID, Value, and Weight
        {0, 10, 5},    // Item 0: Value = 10, Weight = 5
        {1, 60, 10},   // Item 1: Value = 60, Weight = 10
        {2, 100, 20},  // Item 2: Value = 100, Weight = 20
        {3, 120, 30}   // Item 3: Value = 120, Weight = 30
    };
    int n = sizeof(items) / sizeof(items[0]); //item size/count
    int capacity = 50; // Bag capacity

    float maxProfit = knapsackDP(capacity, items, n); //function
call with parameter capacity, itemsARRAY, and n (total item count)
    cout << "Maximum profit: " << maxProfit << endl;
}
/*Problem Description:
We have a set of items, each with an associated value and weight.
The goal is to select a subset of these items to maximize the
total value while ensuring that the
total weight does not exceed a given capacity (in this case,
50).*/
```

output :

```
Selected items (ID): 3 2
Maximum profit: 220
```

GREEDY STRATEGY :

```cpp
#include <iostream>
using namespace std;
float GreedyKS(float c, int T[][3], int i, int n)
{
    if (c <= 0 || i > n) //if the remaining capacity is less than
or equal to 0 or the index is greater than the total number of
items
        return 0;
    if (c < T[i][2]) //if the remaining capacity is less than the
weight of the current item
    {
        float p = c / T[i][2] * T[i][1];
        return p; //P = profit (c/w * v)
    }

    return T[i][1] + GreedyKS(c - T[i][2], T, i + 1, n);/*return
the value of the current item plus the value of the next item
c - T[i][2] is the remaining capacity after selecting the current
item, and i + 1 is  next item's index, n is last index*\
}

int main()
{ //row is item and column is the id, value, weight,
value/weight,n = 3, c = 50
    int T[3][3] = {
    //   0, 1, 2, 3
     // id, v, w, v/w
        {1, 60, 10},   // Item 1: Value = 60, Weight = 10
        {2, 100, 20},  // Item 2: Value = 100, Weight = 20
        {3, 120, 30}   // Item 3: Value = 120, Weight = 30
    };
        // {1, 60, 10, 6},
        // {2, 100, 20, 5},
        // {3, 120, 30, 4}};
    float c = 50;  //C = Bag capacity
```

```
 float profit = GreedyKS(c, T, 0, 3); 0 is the starting index and
3 is the last index

  cout << profit << endl;
    return 0;
}
/*
```

The GreedyKS function implements a greedy algorithm for the knapsack problem.:

It calculates the maximum profit that can be obtained by selecting items with the highest value-to-weight ratio while staying within the given capacity c.if the remaining capacity is less than the weight of the current item, it partially selects the item based on the remaining capacity.The function recursively considers the next item and updates the remaining capacity.The base case is when the remaining capacity is zero or all items have been considered.

* c: A float representing the remaining capacity.

* T[][4]: A 2D array (matrix) representing the items. Each row contains an item's ID, value, weight, and value-to-weight ratio.

* i: An integer representing the current item index.

* n: An integer representing the total number of items.

4. // FUNCTION EXPLANATION :

5. int T[3][4] = 2D array T with three rows and four columns. Each row represents an item, and the columns represent the item's ID, value, weight, and value-to-weight ratio.

7. float profit = GreedyKS(c, T, 0, 2);: This calculates the maximum profit using the GreedyKS function. The initial item index is 0, and there are 3 items (so n is 2).

8. cout << profit << endl;:

This prints the calculated profit The line float profit = GreedyKS(c, T, 0, 3); in the code calls the GreedyKS function to solve the knapsack problem greedily. It passes the knapsack capacity c, the array of items T,the starting index 0 indicating the first item, and 3 as the last index (although it should be 2 for the third item). The function calculates the maximum profit achievable and stores it in the variable profit.

OUTPUT :

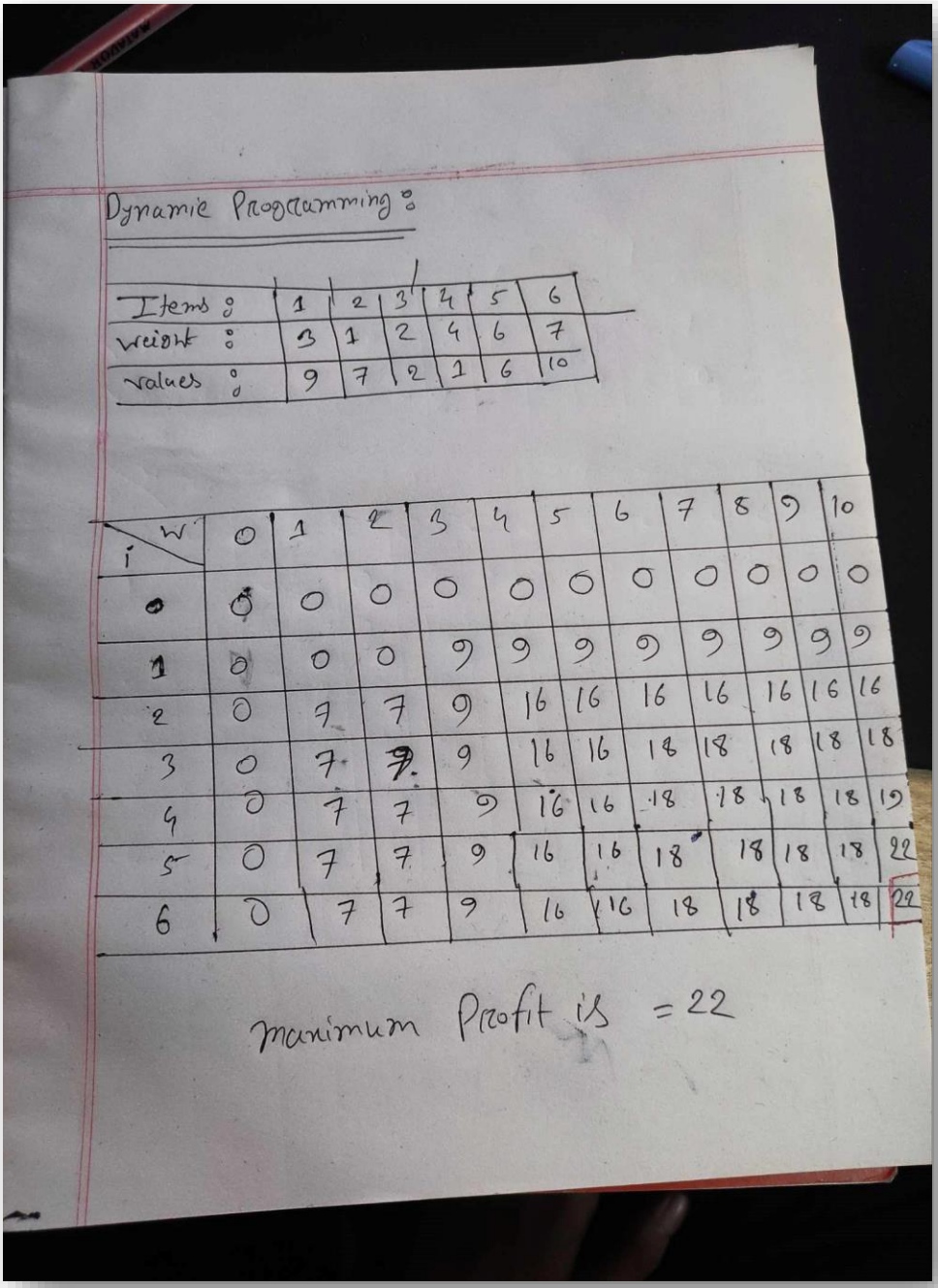
```
Maximum profit: 240
Number of items selected: 3
Items selected (ID): 3 2 1
```

2. ALGORITHM SIMULATION & ANALYSIS :

a. Dynamic Programming (DP):

• Simulate the dynamic programming approach to solve the 01 Knapsack

Problem on at least 6 items and a knapsack capacity of 10.



Analyze the time complexity of the algorithm in the implementation part :

The time complexity for solving the 0/1 knapsack problem using dynamic programming is typically O(nW),

where 'n' is the number of items and 'W' is the capacity of the knapsack.

1. Initialization: Creating a 2D array of size (n+1) x (W+1) to store the maximum value that can be obtained using items up to the 'ith' item and a knapsack capacity of 'j'. This step takes O(nW) time as it involves initializing all elements of the array.
2. Filling the Table: For each item 'i' (from 1 to n) and for each capacity 'j' (from 0 to W), we calculate the maximum value that can be obtained considering whether to include the current item or not. This step also takes O(nW) time since we fill each cell of the 2D array once, and each filling operation takes constant time.

Overall, the time complexity for solving the 0/1 knapsack problem using dynamic programming is O(nW). However, there are optimizations and variations of the problem that can lead to different time complexities. For example, if the weights or values are bounded by a constant, the time complexity might be different. But in the general case, it remains O(nW).

b. *Greedy Strategy:*

Simulate the greedy strategy to solve the 01 Knapsack Problem using the

same data used in DP.

Items :	1	2	3	4	5	6
Weight (w) :	3	1	2	4	6	7
Value (v) :	9	7	2	1	6	10

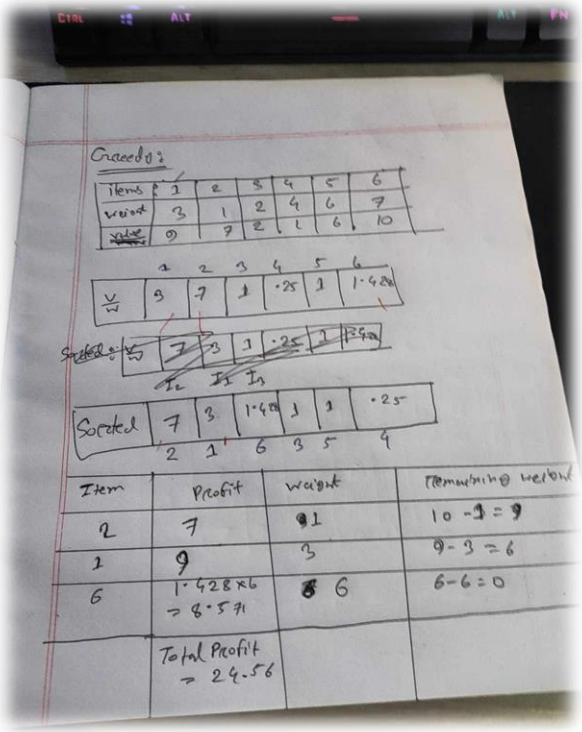
Now (v/w) :

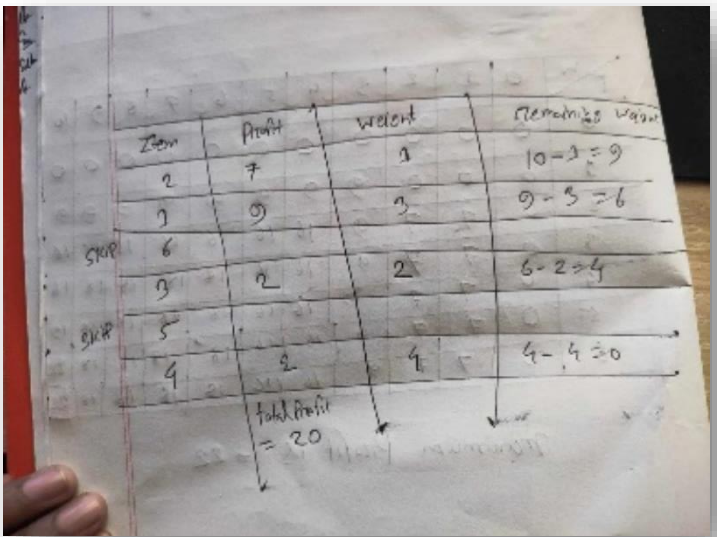
v/w :	3	7	1	.25	1	1.428
Item :	1	2	3	4	5	6

Sorted data :

Item ;	2	1	6	3	5	4
WEIGHT						
VALUE	7	9	7	2	6	1
V/W	7	3	1.428	1	1	.25

Here we cant take item 6 as in
Question it written that we need
To solve 0/1 knapsack using greedy
method.and we cant take fractional
value item 6 has fractional value.
so the answer has to be **7+9 = 16,**
But we can skip these fractional value and jump to next value 7+9+2+2 = 20.
which is also not optimal . but in Dynamic programming we are getting **22** as
profit which is optimal.





Analyze the time complexity of the algorithm in the implementation part:

1. **Greedy Algorithm for 0/1 Knapsack:**
    - The provided code aims to solve the 0/1 Knapsack Problem using a greedy approach.
    - However, it's essential to note that the greedy algorithm is not optimal for the 0/1 Knapsack Problem.
    - The greedy strategy (selecting items based on value-to-weight ratio) does not guarantee an optimal solution because it may lead to suboptimal choices.
    - The 0/1 Knapsack Problem requires dynamic programming or other techniques for an optimal solution.
2. **Time Complexity:**
    - The time complexity of the given code depends on the recursive function GreedyKS.
    - time complexity:
        - The function GreedyKS is called recursively.
        - In each recursive call, we either include the current item (if its weight allows) or exclude it.
        - The function explores all possible combinations of items.
        - The total number of recursive calls is proportional to the number of items (denoted by N).
        - For each item, we make two recursive calls (include or exclude).
        - Therefore, the total number of recursive calls is 2^N (exponential time complexity).
        - Additionally, within each recursive call, we perform constant-time operations (comparisons and calculations).
        - Overall, the time complexity is dominated by the exponential factor: O(2^N).
3. **Space Complexity:**
    - The space complexity is determined by the recursive call stack.
    - Since we have N recursive calls, the maximum depth of the call stack is N.
    - Therefore, the space complexity is O(N).
4. **Conclusion:**

- The given code uses a greedy approach, but it is inefficient for the 0/1 Knapsack Problem.
- **T**o solve the problem optimally, consider using dynamic programming (which has a more efficient time complexity of O(N * W), where W is the capacity of the knapsack).

it is not suitable for solving the 0/1 Knapsack Problem optimally due to its exponential time complexity. For practical applications, consider using dynamic programming or other efficient algorithms.

Discuss the advantages and limitations of the greedy approach for this problem.

Advantages of Greedy Algorithm:

The 0/1 Knapsack Problem is a classic optimization problem where we have a set of items, each with a weight and a value, and we want to select a subset of these items to maximize the total value while ensuring that the total weight does not exceed a given capacity.
the time complexity of the greedy algorithm for the 0/1 Knapsack:

1. Greedy Algorithm for Fractional Knapsack:First, let's discuss the greedy algorithm for the Fractional Knapsack problem. In this variant, we can take fractional parts of items to maximize the total value.

o The greedy approach sorts the items by their value-to-weight ratio (value divided by weight) in descending order.

o Then, it iteratively adds items to the knapsack starting from the highest value-to-weight ratio until the capacity is exhausted.

o The time complexity of this greedy algorithm is O(N log N), where N is the number of items1.

2. Greedy Algorithm for 0/1 Knapsack:

o Unfortunately, there is no efficient greedy algorithm for the 0/1 Knapsack problem.

o Unlike the fractional variant, in the 0/1 Knapsack, we must either take an entire item or leave it entirely (no fractional parts allowed).

o Greedy strategies that work for fractional knapsack (like selecting items based on value-to-weight ratio) do not guarantee an optimal solution for the 0/1 Knapsack.

o The reason is that the greedy approach may lead to suboptimal choices, resulting in a solution that is not globally optimal.

o Therefore, greedy algorithms fail to guarantee optimality for the 0/1 Knapsack problem.Solving it optimally requires dynamic programming or other techniques.

o The dynamic programming approach has a time complexity of O(N * W), where N is the number of elements and W is the capacity of the knapsack2.

In summary, while greedy algorithms work well for the fractional knapsack, they are not suitable for the 0/1 Knapsack due to its discrete nature and the need to make binary decisions about item inclusion. For the latter, dynamic programming provides an optimal solution with a more efficient time complexity.