

Knapsack problem-greedy and Dynamic

A COURSE PROJECT REPORT

Submitted by

Raj Kumar Jaiswal (RA2111003011888)

Ritik Singh Pundhir(RA2111002011861)

Under the guidance of

Dr. Geetha R.

In partial fulfilment for the Course

Of

18CSC204J – DESIGN AND ANALYSIS OF ALGORITHMS

in Computing Technologies



FACULTY OF ENGINEERING AND TECHNOLOGY SRM

INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chenpalpattu District

May 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this Course Project Report Titled “Knapsack problem-greedy and dynamic” is the bonafide work done by **Ritik Singh Pundhir(RA2111002011861)** **Raj Kumar Jaiswal (RA2111003011888)** who carried out the project work under my supervision. Certified further, that do the best of my knowledge the work reported here in does not form part of any other work.

SIGNATURE

Faculty In-Charge
Dr. Geetha R.
Assistant Professor,
Department of computing
Technologies
S.R.M. institute of science
and technology

SIGNATURE

HEAD OF THE DEPARTMENT
Dr. M. Pushpalatha
Professor and Head Of Computing
Technologies
S.R.M. institute of science and
technology.

TABLE OF CONTENTS:

1) INTRODUCTION TO GREEDY ALGORITHM

2) INTRODUCTION TO DYNAMIC ALGORITHM

3) KNAPSACK PROBLEM –GREEDY ALGORITHM:

- CODE
- DRY RUN
- TIME COMPLEXITY

4) KNAPSACK PROBLEM –DYNAMIC ALGORITHM:

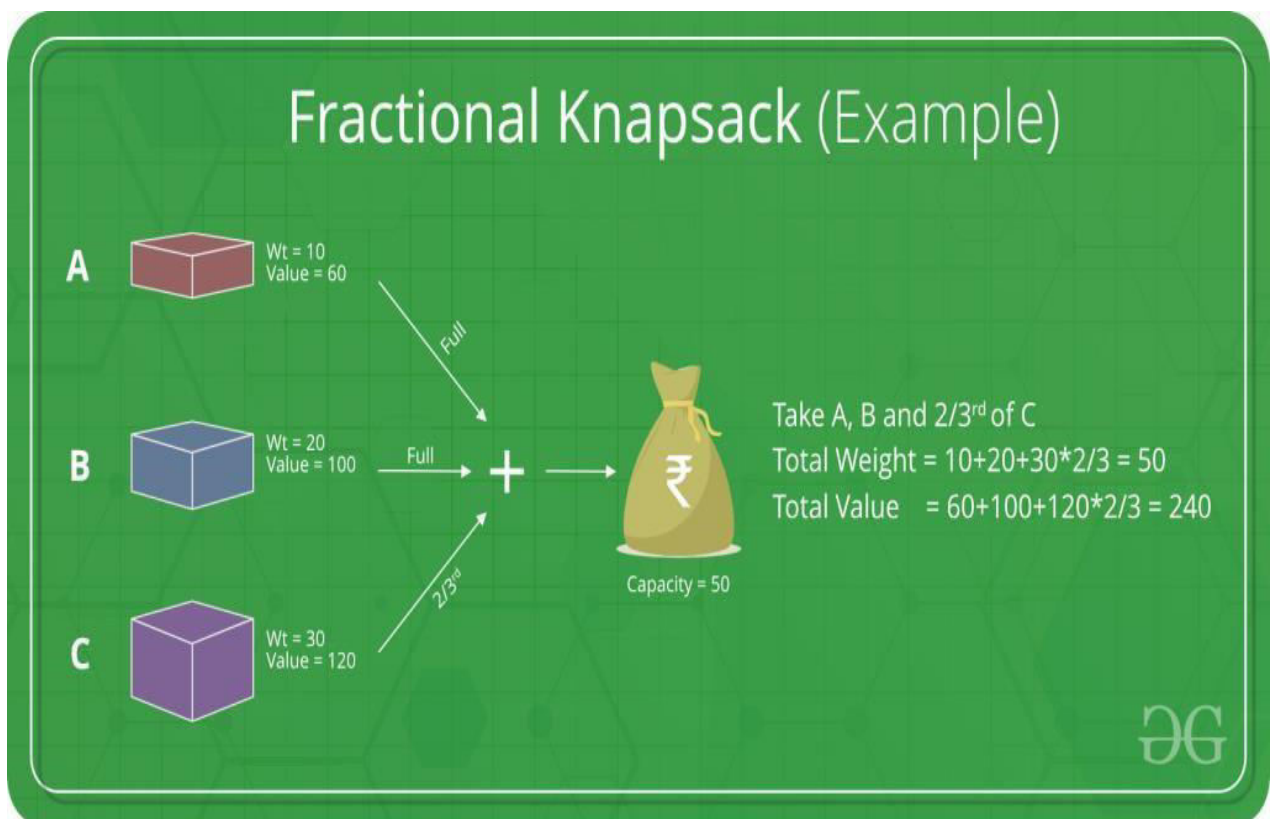
- CODE
- DRY RUN
- TIME COMPLEXITY

5) COMPARATIVE STUDY CONCLUSION

INTRODUCTION TO GREEDY ALGORITHM :

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solutions are the best fit for Greedy.

For example, consider the Fractional Knapsack Problem. The local optimal strategy is to choose the item that has the maximum value vs. weight ratio. This strategy also leads to a globally optimal solution because we are allowed to take fractions of an item.



INTRODUCTION TO DYNAMIC ALGORITHM :

Dynamic Programming is mainly an optimization over plain recursion.

Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of sub-problems so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

For example, if we write a simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of sub-problems, time complexity reduces to linear.

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Recursion : Exponential

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

Dynamic Programming : Linear



1. KNAPSACK PROBLEM – GREEDY ALGORITHM:

The basic idea of the greedy approach is to calculate the ratio profit/weight for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add as much as we can (can be the whole element or a fraction of it).

This will always give the maximum profit because, in each step, it adds an element such that this is the maximum possible profit for that much weight.

Follow the given steps to solve the problem using the above approach:

- Calculate the ratio (profit/weight) for each item.
- Sort all the items in decreasing order of the ratio.
- Initialize $res = 0$, $curr_cap = \text{given cap}$.
- Do the following for every item i in the sorted order:
 - If the weight of the current item is less than or equal to the remaining capacity, then add the value of that item to the result
 - Else add the current item as much as we can and break out of the loop.
- Return res .

1.1 CODE :

```
// C++ program to solve fractional Knapsack Problem

#include <bits/stdc++.h>
using namespace std;

// Structure for an item which stores weight and
// corresponding value of Item
struct Item {
    int profit, weight;

    // Constructor
    Item(int profit, int weight)
    {
        this->profit = profit;
        this->weight = weight;
    }
};

// Comparison function to sort Item
// according to profit/weight ratio
static bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.profit / (double)a.weight;
    double r2 = (double)b.profit / (double)b.weight;
    return r1 > r2;
}

// Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int N)
{
    // Sorting Item on basis of ratio
    sort(arr, arr + N, cmp);

    double finalvalue = 0.0;

    // Looping through all items
    for (int i = 0; i < N; i++) {

        // If adding Item won't overflow,
        // add it completely
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
            finalvalue += arr[i].profit;
        }
    }
}
```

```

        // If we can't add current Item,
        // add fractional part of it
        else {
            finalvalue
                += arr[i].profit
                * ((double)W / (double)arr[i].weight);
            break;
        }
    }

    // Returning final value
    return finalvalue;
}

// Driver code
int main()
{
    int W = 50;
    Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    cout << fractionalKnapsack(W, arr, N);
    return 0;
}

```

```

main.cpp
1 #include <bits/stdc++.h>
2 using namespace std;
3 struct Item {
4     int profit, weight;
5     Item(int profit, int weight)
6     {
7         this->profit = profit;
8         this->weight = weight;
9     }
10 };
11 static bool cmp(struct Item a, struct Item b)
12 {
13     double r1 = (double)a.profit / (double)a.weight;
14     double r2 = (double)b.profit / (double)b.weight;
15     return r1 > r2;
16 }
17 double fractionalKnapsack(int W, struct Item arr[], int N)
18 {
19     sort(arr, arr + N, cmp);
20     double finalvalue = 0.0;
21     for (int i = 0; i < N; i++) {
22         if (arr[i].weight <= W) {
23             W -= arr[i].weight;
24             finalvalue += arr[i].profit;
25         }
26         else {
27             finalvalue
28                 += arr[i].profit
29                 * ((double)W / (double)arr[i].weight);
30             break;
31         }
32     }
33     return finalvalue;
34 }
35 int main()
36 {
37     int W = 50;
38     Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
39     int N = sizeof(arr) / sizeof(arr[0]);
40     cout << fractionalKnapsack(W, arr, N);
41     return 0;
42 }

```

Output

```

/tmp/EXc080G5iy.o
240

```


1.2 DRY RUN :

Given the weights and profits of **N** items, in the form of **{profit, weight}** put these items in a knapsack of capacity **W** to get the maximum total profit in the knapsack. In **Fractional Knapsack**, we can break items for maximizing the total value of the knapsack.

Input: arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50

Output: 240

Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg. Hence total price will be $60+100+(2/3)(120) = 240$

Input: arr[] = {{500, 30}}, W = 10 **Output:**

166.667

Consider the example: **arr[] = {{100, 20}, {60, 10}, {120, 30}}, W = 50.**

Sorting: Initially sort the array based on the profit/weight ratio. The sorted array will be {{60, 10}, {100, 20}, {120, 30}}.

1.3 TIME COMPLEXITY :

Time Complexity: $O(2^N)$

Auxiliary Space: $O(N)$

2. KNAPSACK PROBLEM – DYNAMIC PROBLEM :

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W . From all such subsets, pick the subset with the maximum profit.

Optimal Substructure: To consider all subsets of items, there can be two cases for every item.

- **Case 1:** The item is included in the optimal subset.
- **Case 2:** The item is not included in the optimal set.

We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W , [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

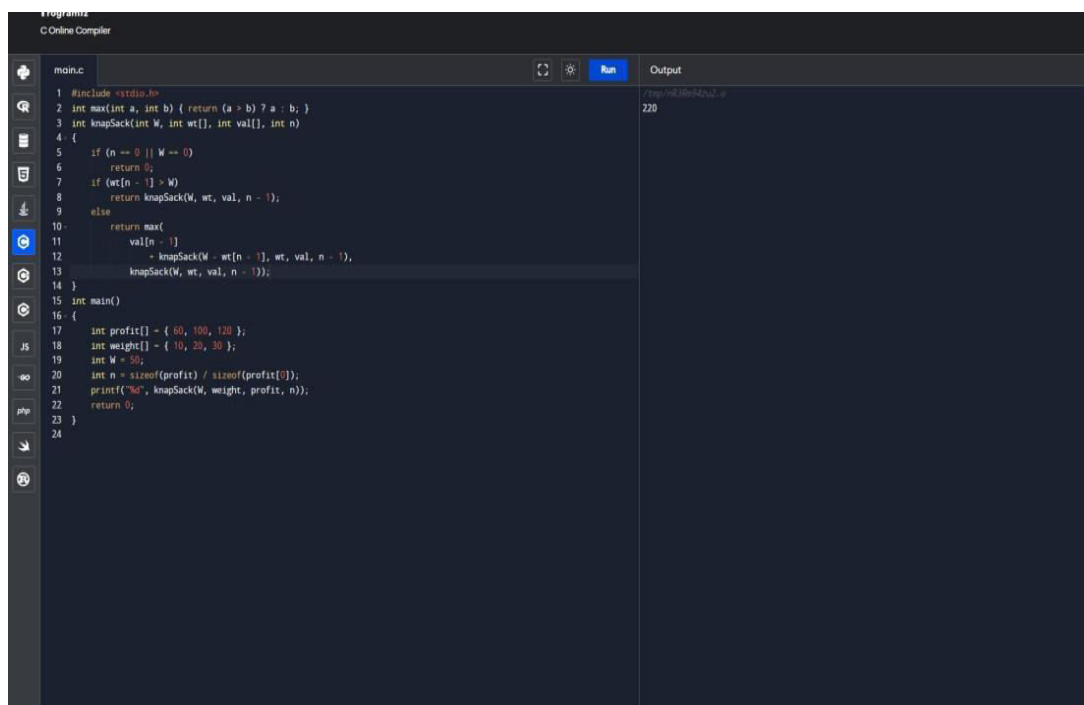
Follow the below steps to solve the problem:

The maximum value obtained from ' N ' items is the max of the following two values.

- Maximum value obtained by $N-1$ items and W weight (excluding n^{th} item)
 - Value of n^{th} item plus maximum value obtained by $N-1$ items and $(W - \text{the weight of the } N^{\text{th}} \text{ item})$ [including N^{th} item].
- If the weight of the ' N^{th} ' item is greater than ' W ', then the N^{th} item cannot be included and **Case 1** is the only possibility.

2.1 CODE :

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}
int
main() {
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("%d", knapSack(W, weight, profit, n));
    return 0;
}
```



The screenshot shows a web-based C compiler interface. On the left, there is a sidebar with icons for file management and a list of languages (C, JS, PHP, Python). The main editor area displays the C code for the knapSack problem, with line numbers 1 through 24. The code is identical to the one provided in the previous block. On the right, there is an 'Output' panel showing the result of the program execution, which is the number 220. The 'Run' button is highlighted in blue.

2.2 DRY RUN

Input: $N = 3$, $W = 4$, $\text{profit}[] = \{1, 2, 3\}$, $\text{weight}[] = \{4, 5, 1\}$ **Output:**

3

Explanation: There are two items that have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weights 4 and 1 together as the capacity of the bag is 4.

Input: $N = 3$, $W = 3$, $\text{profit}[] = \{1, 2, 3\}$, $\text{weight}[] = \{4, 5, 6\}$ **Output:**

0

Let, $\text{weight}[] = \{1, 2, 3\}$, $\text{profit}[] = \{10, 15, 40\}$, $\text{Capacity} = 6$ •

If no element is filled, then the possible profit is 0.

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
weight→ item↓/	0	1	2	3	4	5	6
1							
2							
3							

- **For filling the first item in the bag:** If we follow the above-mentioned procedure, the table will look like the following.

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10

2							
3							

- **For filling the second item:**

When $j = 2$, then the maximum possible profit is $\max(10, DP[1][2-2] + 15) = \max(10, 15) = 15$.

When $j = 3$, then maximum possible profit is $\max(2 \text{ not put}, 2 \text{ is put into bag}) = \max(DP[1][3], 15 + DP[1][3-2]) = \max(10, 25) = 25$.

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3							

- **For filling the third item:**

When $j = 3$, the maximum possible profit is $\max(DP[2][3], 40 + DP[2][3-3]) = \max(25, 40) = 40$.

When $j = 4$, the maximum possible profit is $\max(DP[2][4], 40 + DP[2][4-3]) = \max(25, 50) = 50$.

When $j = 5$, the maximum possible profit is $\max(DP[2][5], 40 + DP[2][5-3]) = \max(25, 55) = 55$.

When $j = 6$, the maximum possible profit is $\max(DP[2][6], 40 + DP[2][6-3]) = \max(25, 65) = 65$.

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0

1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

2.3 TIME COMPLEXITY

Time Complexity: $O(N * W)$. where 'N' is the number of elements and 'W' is capacity.

Auxiliary Space: $O(N * W)$. The use of a 2-D array of size 'N*W'.

COMPARATIVE STUDY:

There are two main approaches to solving the knapsack problem: greedy algorithms and dynamic programming. Here is a comparative study of these two approaches:

1. Greedy Algorithm:

The greedy algorithm is a simple, intuitive algorithm that makes locally optimal choices at each step in the hope of finding a global optimum. The basic idea is to sort the items by their value-to-weight ratio and then add items to the knapsack one by one, starting with the most valuable items until the knapsack is full. The time complexity of the greedy algorithm is $O(n \log n)$ if we use a sorting algorithm to sort the items by their value-to-weight ratio.

Advantages:

- The greedy algorithm is simple and easy to implement.
- It has a low time complexity, which makes it efficient for small instances of the problem.

Disadvantages:

- The greedy algorithm does not always produce an optimal solution. There may be cases where it chooses a locally optimal item that leads to a suboptimal overall solution.

2. Dynamic Programming:

Dynamic programming is a more complex algorithmic approach that breaks down the problem into smaller sub-problems and uses the optimal solutions to these subproblems to construct the optimal solution to the original problem. The basic idea is to create a table that stores the optimal values for each sub-problem and use these values to calculate the optimal solution to the original problem. The time complexity of dynamic programming is $O(nW)$, where n is the number of items and W is the capacity of the knapsack.

Advantages:

- Dynamic programming always produces an optimal solution.
- It can handle larger instances of the problem.

Disadvantages:

- Dynamic programming has a higher time complexity than the greedy algorithm, which makes it less efficient for small instances of the problem.
- It can be more difficult to implement than the greedy algorithm.

CONCLUSION:

Both the greedy algorithm and dynamic programming can be used to solve the knapsack problem, but they have different strengths and weaknesses. The greedy algorithm is simple and efficient for small instances of the problem, but it may not always produce an optimal solution. Dynamic programming is more complex and has a higher time complexity, but it always produces an optimal solution and can handle larger instances of the problem. The choice between these two approaches depends on the specific requirements of the problem and the available computing resources.