

DEPARTMENT OF NETWORKING AND COMMUNICATION

0/1 KNAPSACK PROBLEM

PROJECT REPORT

Academic Year: 2022-23

EVEN SEMSTER

Program : UG
Semester : IV
Course Code : 18CSC204J
Course Title : DESIGN AND ANALYSIS OF ALGORITHMS
Student Name : B V N RAHUL
I SAI SRI TEJA
K VINEETH
Register Number : RA2111028010192
RA2111028010170
RA2111028010179
Branch with Specialization : CSE – CLOUD COMPUTING
Section : Q1



SCHOOL OF COMPUTING

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur – 603 203

BONAFIDE

This is to certify that **18CSC204J - DESIGN AND ANALYSIS OF ALGORITHMS** Mini Project report titled “**0/1 Knapsack Problem**” is the bonafide work of **B V N Rahul (RA2111028010192), I Sai Sri Teja (RA2111028010170), K Vineeth (RA2111028010170)** and who undertook the task of completing the project within the allotted time.

Signature

Signature

Mr. Rajasoundaran S

Assistant Professor

Department of Networking and
Communication

SRM Institute of Science and
Technology

Dr. Annapurani Panaiyappan K

Professor and Head

Department of Networking and
Communication

SRM Institute of Science and
Technology

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable Vice Chancellor **Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy**, for his encouragement

We express our profound gratitude to our Dean (College of Engineering and Technology) **Dr. T. V.Gopal**, for bringing out novelty in all executions.

We would like to express my heartfelt thanks to Chairperson, School of Computing Dr. Revathi Venkataraman, for imparting confidence to complete my course project

We wish to express my sincere thanks to Course Audit **Professor Dr.Annapurani Panaiyappan**, Professor and Head, Department of Networking and Communications and Course Coordinators for their constant encouragement and support.

We are highly thankful to our my Course project Faculty **Rajasoundaran S** Assistant Professor for his/her assistance, timely suggestion and guidance throughout the duration of this course project.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

TABLE OF CONTENTS

Chapter No.	Title	Page No.
1	Abstract	5
2	Introduction	6
3	Problem Statement	7
4	Problem Definition	8
5	Recursion	8
6	Example for Recursion	9
7	Implementation of code for Recursion	10
8	Dynamic Programming	11
9	Example for Dynamic Programming	11
10	Implementation of code for Dynamic Programming	13
11	Conclusion	14

ABSTRACT

The 0/1 knapsack problem is a classical combinatorial optimization problem that has wide-ranging applications in various domains, such as finance, manufacturing, and logistics. It involves selecting a subset of items with different values and weights to fit into a knapsack with limited capacity, such that the total value of the selected items is maximized. In this project, we aim to explore different algorithms for solving the 0/1 knapsack problem and evaluate their performance using various metrics.

We will start by implementing the brute-force approach and dynamic programming techniques to solve the problem. The brute-force approach involves generating all possible subsets of items and selecting the best solution. Although this method guarantees an optimal solution, it is computationally expensive and impractical for large problem instances. Dynamic programming techniques can exploit the overlapping subproblems and optimal substructure properties of the problem to reduce the computational complexity. We will compare the time and space complexity of these algorithms and analyze their strengths and limitations.

Next, we will experiment with heuristic algorithms such as genetic algorithms and simulated annealing to find good solutions in a reasonable time. These algorithms are based on natural phenomena and can efficiently explore the solution space to find good solutions. We will design and implement these algorithms and evaluate their performance using different parameters and settings.

Finally, we will analyze the scalability and robustness of the algorithms by testing them on different problem instances with varying sizes and characteristics. We will generate random instances and instances with specific properties such as high correlation between value and weight, multiple knapsacks, and bounded knapsacks. We will measure the performance of the algorithms in terms of the solution quality, computation time, and memory usage, and analyze the trade-offs and challenges of each algorithm. The results of this project can provide insights into the effectiveness and trade-offs of different techniques for solving the 0/1 knapsack problem and contribute to the development of efficient optimization methods for real-world applications.

Introduction:

The 0/1 Knapsack Problem is a well-known optimization problem in computer science and operations research. It is a combinatorial problem that involves selecting a subset of items to maximize the total value while satisfying a weight constraint. The problem is defined as follows: we are given a set of n items, each with a weight and a value, and a knapsack with a maximum weight capacity W . The goal is to select a subset of items such that the total weight does not exceed W , and the total value is maximized.

The 0/1 Knapsack Problem is called "0/1" because each item can be either included or excluded from the knapsack, but not partially included. This restriction makes the problem more difficult than the unbounded knapsack problem, where each item can be included multiple times. The 0/1 Knapsack Problem is NP-complete, which means that no polynomial-time algorithm exists that can solve all instances of the problem.

The 0/1 Knapsack Problem has many practical applications in various fields. In manufacturing, it can be used to optimize the selection of items for production, considering their weights and values. In finance, it can be used to optimize the selection of investments, considering their risks and returns. In resource allocation, it can be used to optimize the selection of resources for a project, considering their costs and benefits.

There are various algorithms to solve the 0/1 Knapsack Problem, ranging from simple brute-force approaches to more sophisticated dynamic programming and heuristic algorithms. One of the most used algorithms is the dynamic programming approach, which involves building a table of subproblems and solving each subproblem by considering the optimal solution for the previous subproblems. This approach has a time complexity of $O(nW)$, where n is the number of items and W is the maximum weight capacity of the knapsack.

Another approach is the branch and bound algorithm, which involves generating all possible solutions by creating a tree of possibilities and using lower bounds to prune the branches that cannot lead to the optimal solution. This approach can be more efficient than the dynamic programming approach for larger instances of the problem.

There are also heuristic algorithms, such as the genetic algorithm and the simulated annealing algorithm, which use probabilistic techniques to find good solutions to the problem. These algorithms do not guarantee an optimal solution but can often find good solutions in a reasonable amount of time.

The 0/1 Knapsack Problem is a classic optimization problem that has many practical applications in various fields. Although it is NP-complete, there are many algorithms that can be used to solve the problem efficiently, depending on the size of the problem and the desired level of accuracy. The dynamic programming approach is the most used algorithm, but there are also other approaches, such as the branch and bound algorithm and heuristic algorithms, that can be used to solve the problem.

Problem Statement:

Given a set of n items, where each item i has a weight w_i and a value v_i , and a knapsack of maximum weight capacity W , the objective is to find a subset S of the items such that the total weight of the items in S does not exceed W , and the total value of the items in S is maximized. In other words, we want to find a binary vector x of length n such that x_i is either 0 or 1 (meaning that the i -th item is not selected or selected, respectively), and the following conditions are satisfied:

$\sum_{i=1}^n x_i w_i \leq W$ (the total weight of the selected items does not exceed the capacity of the knapsack)

maximize $\sum_{i=1}^n x_i v_i$ (the total value of the selected items is maximized)

Note that the problem is called the "0/1" Knapsack Problem because each item can either be included in the knapsack or not, but not partially. This makes the problem more difficult than the unbounded knapsack problem, where each item can be included an unlimited number of times.

Problem Definition:

The 0/1 Knapsack problem is a classic optimization problem in computer science, where you are given a set of items with weights and values, and a knapsack with a certain capacity. The goal is to choose a subset of items that maximizes the value while keeping the total weight within the capacity of the knapsack.

Approaches:

There are different approaches to solve the 0/1 knapsack problem, some of them include:

1.Recursion:

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 maximum profit.

Algorithm:

Inputs:

capacity: the maximum weight capacity of the knapsack

weights: a list of weights for each item

values: a list of values for each item

n: the number of items in the list

Output:

The maximum value that can be obtained by filling the knapsack

Step 1: Start

Step 2: Define a recursive function knapsack that takes capacity, weights, values, and n as input. If n is zero or capacity is zero, return 0.

Step 3: If the weight of the last item in the list is greater than capacity, call the knapsack function recursively with n decremented by 1 and return the result.

Step 4: If the weight of the last item in the list is not greater than capacity, return the maximum value of the following two cases:

- i. Include the last item in the list and call the knapsack function recursively with n decremented by 1 and capacity decreased by the weight of the last item.

- ii. Exclude the last item in the list and call the knapsack function recursively with n decremented by 1 and capacity unchanged.

Step 5: Display the maximum value obtained from step 3 and step 4.

Step 6: Stop

Example:

Suppose you are a thief and you have a knapsack that can hold a maximum weight of 8 kg. You have the following items in front of you:

Item	Weight(kg)	Value(\$)
A	1	2
B	3	4
C	5	7
D	7	10

To solve this problem recursively, we can consider the following approach.

Start with the last item (D), and recursively consider two cases:

Case 1: Do not include item D in the knapsack. In this case, the problem reduces to finding the maximum value of items A, B and C with a maximum knapsack capacity of 8 kg (since we have already excluded item D).

Case 2: Include item D in the knapsack. In this case, the problem reduces to finding the maximum value of items A, B and C with a maximum knapsack capacity of 7 kg (since we have already used 7 kg capacity for item D).

Repeat the above steps recursively for the remaining items (A, B and C) until all items have been considered.

Return the maximum value obtained from all the recursive calls.

This means that the maximum value you can steal is 12, by stealing items D and A.


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[] = { 2,4,7,10 };
    int weight[] = { 1,3,5,7 };
    int W = 8;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("%d", knapSack(W, weight, profit, n));
    return 0;
}
```

main.c	Run	Output
<pre>1 #include <stdio.h> 2 int max(int a, int b) { return (a > b) ? a : b; } 3 int knapSack(int W, int wt[], int val[], int n) 4 { 5 if (n == 0 W == 0) 6 return 0; 7 if (wt[n - 1] > W) 8 return knapSack(W, wt, val, n - 1); 9 else 10 return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1), 11 knapSack(W, wt, val, n - 1)); 12 } 13 int main() 14 { 15 int profit[] = { 2,4,7,10 }; 16 int weight[] = { 1,3,5,7 }; 17 int W = 8; 18 int n = sizeof(profit) / sizeof(profit[0]); 19 printf("%d", knapSack(W, weight, profit, n)); 20 return 0; 21 }</pre>		<pre>/tmp/NAUx0GEbeV.o 12</pre>

Time Complexity: $O(2^n)$

2. Dynamic Programming:

In the above approach we can observe that we are calling recursion for same sub problems again and again thus resulting in overlapping subproblems thus we can make use of Dynamic programming to solve 0/1 Knapsack problem.

Algorithm:

- Step 1: Consider the same cases as mentioned in the recursive approach.
- Step 2: In a DP [][] table let's consider all the possible weights from '1' to 'W' as the columns and the element that can be kept as rows.
- Step 3: The state $DP[i][j]$ will denote the maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi'(weight in 'ith' row) we can fill it in all columns which have 'weight values $> w_i$ '. Now two possibilities can take place.
- Step 4: Fill 'wi' in the given column.
- Step 5: Do not fill 'wi' in the given column.
- Step 6: Now we have to take a maximum of these two possibilities,
- Step 7: Formally if we do not fill the 'ith' weight in the 'jth' column then the $DP[i][j]$ state will be the same as $DP[i-1][j]$
- Step 8: But if we fill the weight, $DP[i][j]$ will be equal to the value of ('wi'+ value of the column weighing 'j-wi') in the previous row.
- Step 9: So, we take the maximum of these two possibilities to fill the current state.
- Step 10: End

Example:

Consider the same above example

Item	Weight(kg)	Value(\$)
A	1	2
B	3	4
C	5	7
D	7	10

Construct an adjacency table with maximum weight of knapsack as rows and items with respective weights and profits as columns.

Values to be stored in the table are cumulative profits of the items whose weights do not exceed the maximum weight of the knapsack (designated values of each row).

So, we add zeroes to the 0th row and 0th column because if the weight of item is 0, then it weighs nothing; if the maximum weight of knapsack is 0, then no item can be added into the knapsack.

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1(1,2)	0								
2(3,4)	0								
3(5,7)	0								
4(7,10)	0								

The formula to store the profit values is $\Rightarrow c[i,w]=\max\{c[i-1,w-w[i]]+P[i]\}$

By computing all the values using the formula, the table obtained would be –

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1(1,2)	0	1	1	1	1	1	1	1	1
2(3,4)	0	1	1	4	6	6	6	6	6
3(5,7)	0	1	1	4	6	7	9	9	11
4(7,10)	0	1	1	4	6	9	9	10	12

To find the items to be added in the knapsack, recognize the maximum profit from the table and identify the items that make up the profit, in this example, its $\{1, 7\}$.

The optimal solution is $\{1, 7\}$ with the maximum profit is 12

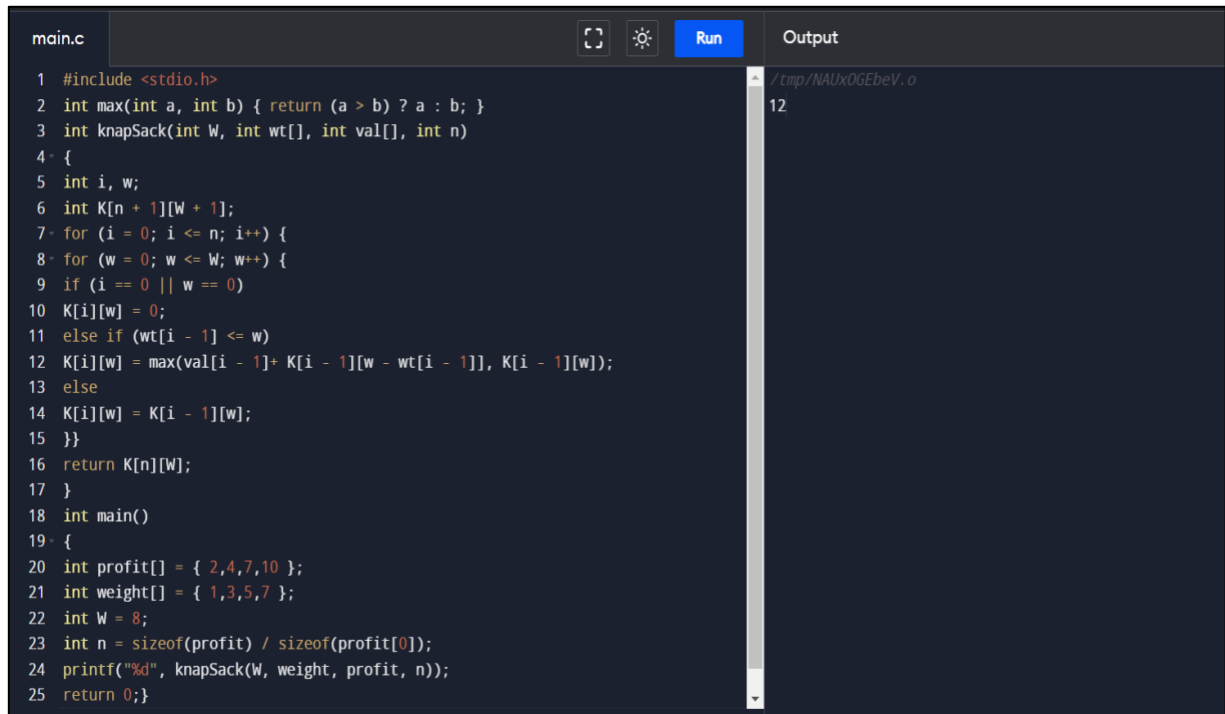
Code:

```
#include <stdio.h>

int max(int a, int b) { return (a > b) ? a : b; }

int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}

int main()
{
    int profit[] = { 2,4,7,10 };
    int weight[] = { 1,3,5,7 };
    int W = 8;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("%d", knapSack(W, weight, profit, n));
    return 0;
}
```



```
main.c
1 #include <stdio.h>
2 int max(int a, int b) { return (a > b) ? a : b; }
3 int knapSack(int W, int wt[], int val[], int n)
4 {
5     int i, w;
6     int K[n + 1][W + 1];
7     for (i = 0; i <= n; i++) {
8         for (w = 0; w <= W; w++) {
9             if (i == 0 || w == 0)
10                K[i][w] = 0;
11             else if (wt[i - 1] <= w)
12                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
13             else
14                K[i][w] = K[i - 1][w];
15         }
16     }
17     return K[n][W];
18 }
19 int main()
20 {
21     int profit[] = { 2,4,7,10 };
22     int weight[] = { 1,3,5,7 };
23     int W = 8;
24     int n = sizeof(profit) / sizeof(profit[0]);
25     printf("%d", knapSack(W, weight, profit, n));
26     return 0;
}
```

Output: 12

Time Complexity: $O(N * W)$

Conclusion:

Among these approaches Dynamic Programming is the best approach because of the following reasons

Time Complexity: Dynamic programming can often solve problems with much lower time complexity than recursive approaches. This is because dynamic programming reuses previously computed solutions to subproblems, whereas recursive approaches may repeatedly compute the same subproblems over and over again.

Space Complexity: Recursive approaches often use a lot of memory due to the large number of functions calls on the stack. Dynamic programming, on the other hand, only needs to store solutions to subproblems, which can often be done in an efficient data structure such as a table or an array.

Flexibility: Dynamic programming is often more flexible than recursion in terms of the types of problems it can solve. Dynamic programming can be applied to a wide range of problems, whereas recursive approaches may not always be applicable.

Ease of implementation: Dynamic programming is often easier to implement than recursive approaches. Recursive algorithms can be more difficult to code and debug because they often require careful management of the call stack, while dynamic programming algorithms can be implemented using simple loops and tables.

Optimization: Dynamic programming can be easily optimized to further reduce time and space complexity, whereas recursive approaches may require more complex optimizations.