Design and Analysis of Algorithms

Group-2

Nidhi kamewar — IIT2019189

Rishi Mukesh Gupta — IIT2019190

Pechetti Venkata Karthik — IIT2019191

Abstract:-Given weights and values of n items we have to design an algorithm to find out the maximum value subset of the values of final items which must be included in the knapsack such that the total value of the weight of the final items included in the knapsack should be less than or equal to the given capacity of the knapsack

I. INTRODUCTION

In this problem,we have to maximise the value of the items included in the knapsack such that the weight of the items included does not exceed the given weight .
As the given problem has Overlapping subproblems of recalculating the total value sum of the items included in the knapsack for a given limit of the knapsack and Optimal substructure property i.e the given problem can be solved by solving the subproblems of the problem

Hence We can solve the given problem using Dynamic programming.

This report further contains -
II. Algorithm Design
III. Algorithm Analysis
IV. Experimental Study
V. Conclusion

II.ALGORITHM DESIGN

Algorithm-1:

1.We have to check for all possible valid subsets of the given set of values and respective weights

2.There are two possible cases for every item either to be included or not to be included in the subset .

3.By considering both of the cases we choose the one which gives us the maximum value of the items for a given capacity of the knapsack.

4.The function recursively calls itself to find the maximum value of the items excluding the current item and including the current item.

5.If the weight of the current item is greater than the given capacity of the knapsack then we remove that item from the knapsack then we call the function for the remaining elements .

Algorithm-2:

As the first algorithm is not very efficient as it recalculate same result we can remove the work of calculating the same result again by storing the value of the subproblems
So we designed an algorithm based on dynamic programming Tabular(Bottom-Up) Approach

we reach our solution by considering all the subsolutions and storing the solutions to the subproblems in a lookup table which are filled iteratively until we reach the solution to the given problem

1.First, the array of values of the given items and weights of respective items are passed to the function

2.we start at the initial condition of 0 weight limit which implies that the bag cannot be filled and so we fill the lookup table dp[][0] row to all 0's and we fill the values of dp[0][] row to 0's as there will be no items included in the knapsack if there are no items given.

3.The function iteratively fils the values of every cell of the lookup table,If the weight of the item is greater than the current knapsack limit the we cannot add the item into the knapsack so we leave the item i.e, we store the value of dp[i-1][j] in the cell dp[i][j].

4.If the weight of the item is less than the current knapsack limit then we fill the cell with the maximum of values attained by including the item in the knapsack(dp[i-1][j-wt[i-1]]+val[i-1]) and attained by not including the item in the knapsack (dp[i-1][j])

5.Finally we return the value of dp[n][cap] which is the maximum value of the items which can be included in the knapsack.

For Example:-

Consider the following

Values	**Weights**
3	6
9	4
2	6
1	1
6	4

Lookup table:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	3	3
2	0	0	0	0	9	9	9	9
3	0	0	0	0	9	9	9	9
4	0	1	1	1	9	10	10	10
5	0	1	1	1	9	10	10	10

By maintaining the lookup table we removed the task of recalculating the overlapping subproblems.

For example in the above problem we the value of dp[1][3] is not recalculated but instead stored at the first calculation and stored in the lookup table and is directly used in other subproblem .
The algorithm builds the above lookup table iteratively till the last cell (dp[5][7]) which is the solution to our problem and finally returns the value present in the last cell.

PSEUDO CODE

Algorithm-1

```
function Solve_knapsack( )

if W==0 or n==0
        return 0

else if wt[n-1]>Total_weight
        return solve_knapsack(val,wt,W,n-1)

else
return(max
(solve_knapsack(val,wt,W,n-1),solve_knapsack(val,w
t,W-wt[n-1],n-1
+val[n-1]))
end if
```

Algorithm-2

```
function Solve_knapsack( )
for  i ← 0 to n do
 for j ← i to cap do

   if(i==0||j==0)
        dp[i][j]← 0

   else if(wt[i-1]>j)
        dp[i][j]← dp[i-1][j]

   else
dp[i][j]← max(dp[i-1][j],dp[i-1][j-wt[i-1]]+val[i-1])
  end if
```

III. ALGORITHM ANALYSIS

Algorithm-1

Time complexity

The algorithm -1 computes every possible subset to find the solution to the main problem hence it take **O(2^n)** time as the possible cases for an item is either to be include or not be included

Space complexity

The algorithm uses **O(1)** auxiliary space as no extra data structures are required

Algorithm-2

Time complexity

The algorithm-2 does not compute the solutions to overlapping subproblems instead looks for them in the lookup table which stores the values of subproblems which occurred before.

The algorithm stores the solutions of every subproblem by bottom up approach by iterating through all possible subproblems i.e,the time complexity of the algorithm would be **O(n*cap).**

Space complexity

The algorithm uses **O(n*cap)** auxiliary space for the lookup table .

IV.EXPERIMENTAL STUDY

1.Algorithm-1

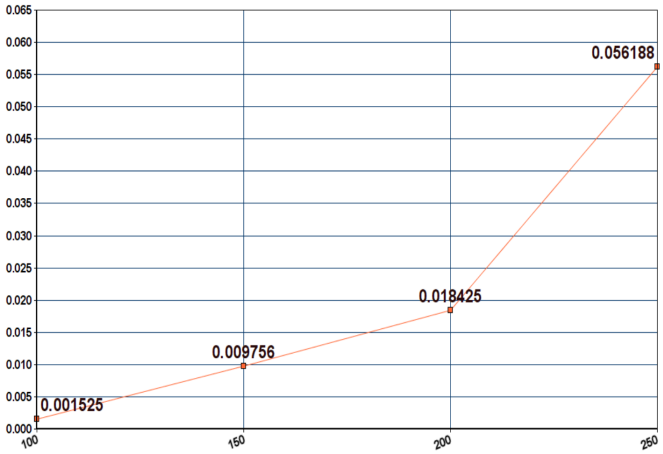

Fig-1 n vs time(Exponential)

n(number of items) X-axis	Time Y-axis
100	0.001525
150	0.009756
200	0.018425
250	0.056188

From the plotted graph (Fig-1) of test cases we can see the time complexity of the algorithm-1 to be **O(2^n)** by asymptotic analysis i.e,the time complexity is exponential

2.Algorithm-2

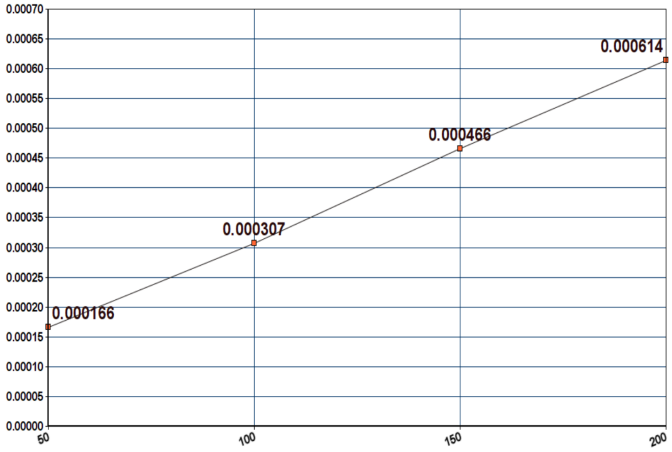

Fig-2 n vs time(linear)

n(number of items) X-axis	capacity	Time Y-axis
50	950	0.000166
100	950	0.000307
150	950	0.000466
200	950	0.000614

Algorithm-2 is a more efficient algorithm as it uses dynamic programming to avoid recalculating the solutions to the overlapping subproblem wheares the algorithm-1 uses exhaustive search by which arrives at the solution by generating all possible subsets .

From the plotted graph(Fig-2) we can see a linear growth of the graph when the capacity of the knapsack is constant as time complexity of algorithm is **O(n*capacity).**

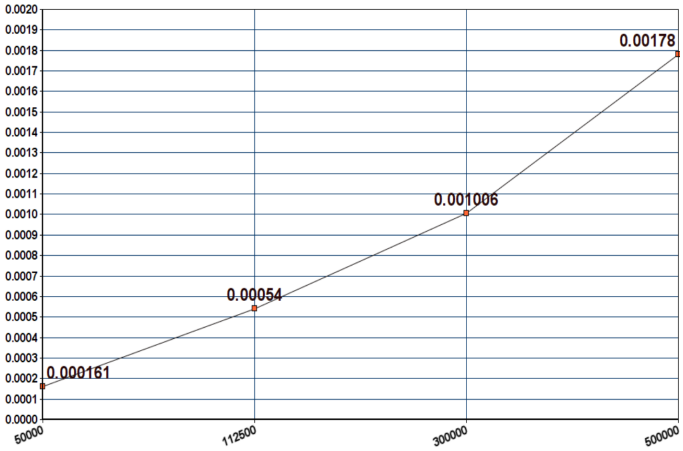

Fig-3 n*capacity vs time

n(number of items) X-axis	Capacity	Time Y-axis
100	500	0.000161
150	750	0.000569
200	1500	0.001006
250	2000	0.001780

From the plotted graph (Fig-3) where capacity is not constant we can see the time complexity of the graph to be **O(n*capacity)**

V.CONCLUSION

From the experimental study we concluded that the running time of the algorithm -2 is better than the first one which can be observed from the study of graphs of algorithm-1 and algorithm-2

REFERENCES

[1] https://www.geeksforgeeks.org
[2] Introduction to Algorithms (CLRS)