Design and Analysis of Algorithms

Group - 2

Group Members:

IIT2019189: Nidhi Kamewar
IIT2019190: Rishi Gupta
IIT2019191: Pechetti Venkata Karthik

Content Listings

- ❑ Problem Statement
- ❑ Introduction
- ❑ Algorithm Design
- ❑ Code Explaination
- ❑ Example
- ❑ Pseudo Code
- ❑ Time Complexity
- ❑ Space Complexity
- ❑ Conclusion
- ❑ References

Problem Statement

Optimal knapsack filling problem

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

Introduction

- In this problem , we have to maximize the value of the items included in the knapsack such that the weight of the items included does not exceed the given weight of the knapsack.
- The given problem has :
  - **Overlapping sub-problems** of recalculating the total value sum of the items included in the knapsack for a given limit of the knapsack.
  - **Optimal substructure property** i.e., the given problem can be solved by solving the sub-problems of the problem.
- Hence, we can solve the given problem using Dynamic programming.

Algorithm - 1

1) We have to check for all possible valid subsets of the given set of values and respective weights.
2) There are 2 possible cases for every item either to be included or not to be included in the subset.
3) By considering both 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

- First algorithm is not that efficient as it recalculates the same result , we can remove the work of recalculating the same result again and again by storing the values of the sub-problems.
- The algorithm we designed is based on **Bottom-Up** approach of Dynamic Programming.
- We reach our solution by considering all the sub-solutions and storing the solutions to the sub problems in a lookup table which is filled iteratively until we reach the solution to the given problem.

Code Explanation

1) First , the array of values of the given items and 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 fills the values of every cell of the lookup table ,if the weight of the item is greater than the current knapsack limit , then we cannot add the item in the knapsack, so we leave it and store 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 and not including the item in the knapsack
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.

Example

Consider the following-

wt[] : 6 4 6 1 4

val[] : 3 9 2 1 6

	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 have reduced the task of recalculating the overlapping sub-problems.
- 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 :

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 , wt,W-wt[n-1],n-1+val[n-1]))

 end if
```

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
```

Time Complexity

- The algorithm-1 computes every possible subsets to find the solution to the main problem hence it takes O(2^n) time.
- The algorithm-2 does not computes the solutions to the overlapping sub-problems instead looks for them in the lookup table which stores the values of sub-problems which occurred before, hence it takes O(n*cap) time.
- Graph:

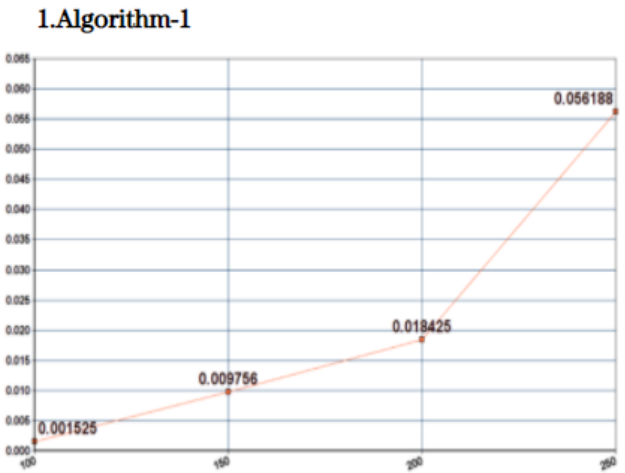

Fig-1 n vs time(Exponential)

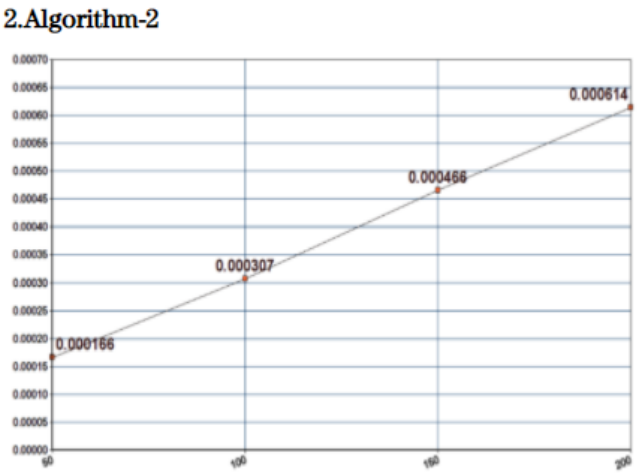

Fig-2 n vs time(linear)

Space Complexity

- The algorithm-1 uses O(1) auxillary space as no extra data structure is required.
- The algorithm-2 uses O(n*cap) auxillary space for the lookup table.
- Graph:

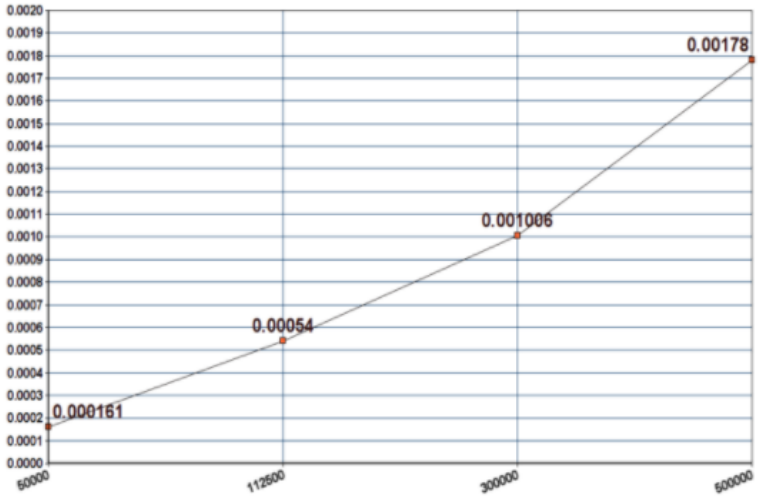

Fig-3 n*capacity vs time

Conclusion

- We proposed an optimal solution to maximize the value of the items included in the knapsack such that weight of the items included does not exceed the given weight.
- The optimal approach has a time complexity of O(n*cap) which is found through the experimental study as well as the asymptotic analysis of the graph.

References

- https://www.geeksforgeeks.org/
- Introduction to Algorithms (CLRS)