Design and Analysis of Algorithms
DAA432C Assignment 06
Group - 14

Kishan Tripathi **IIT2019225**

Mukul Mohmare **IIT2019226**

Anshuman Bharadwaj **IIT2019227**

Abstract :

Illustrate the performance of the 0/1 Knapsack Problem and propose a parallel algorithm for the same.

I. Introduction :

The Knapsack problem is a combinatorial optimization problem where one has to maximize the benefit of objects in a knapsack without exceeding its capacity. Given a set of items we have to find optimal packing of a knapsack. Each item is characterized by weight and value and knapsack is characterized by capacity. Optimal packing is the one in which weight is less or equal to the capacity and in which value is maximal among other feasible packings.

More formally:

Given a number of items n, their weights W = w1......wn, their values V = v1......vn and knapsack capacity c, find vector X = x1.....xn so that (x1*w1 + ...xn*wn) <=c and (x1*w1 + ...xn*wn) is maximal. [1]

II. Algorithm Design :

For each item we are given its weight and its value. We want to maximise the total value of all the items that we are going to put into the knapsack such that the total weight of items is less than or equal to the knapsack's capacity.

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

1. The item is included in the optimal subset,
2. Not included in the optimal set

Therefore, the maximum value that can be obtained from n items is the max of following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If the weight of the nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

Recurrence Relation :
maximum capacity of Knapsack
if (n=0 or W=0)
return 0

if (weight[n] > W)
return solve(n-1, W)
otherwise
return max{ **solve**(n-1, W),
solve(n-1, W-weight[n])

III. Code and Illustration :

If we build the recursion tree for the above relation, we can clearly see that the property of overlapping subproblems is satisfied. So, we will try to solve it using dynamic programming.

Let us define the dp solution with states i and j as

dp[i,j]—> max value that can be obtained with objects u

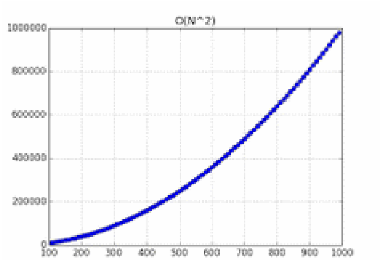
pto index i and knapsack capacity of j.

Code (Top Down Approach) :

```
functionmain()
    val[] = { 60, 100, 120 };
    wt[] = { 10, 20, 30 };
    W = 50;
    n = sizeof(val) / sizeof(val[0]);
    print knapSack(W, wt, val, n);
    return0;

functionknapSackRec(int W, int wt[],int val[], int i,int** dp)
    if (i < 0)
```

```
        then
            return0;
        if (dp[i][W] != -1)
        then
            returndp[i][W];
        if (wt[i] > W)
        then
              dp[i][W] = knapSackRec(W, wt,val, i -
1,dp);
            returndp[i][W];
        else
            dp[i][W] = max(val[i]+knapSackRec(W -
wt[i],
                     wt, val,i - 1, dp),
                       knapSackRec(W, wt, val,i - 1,
dp));
            returndp[i][W];

    functionknapSack(int W, int wt[], int val[], int
n)
        int** dp;
        dp = new int*[n];
        for ( i = 0; i < n; i++)
            dp[i] = new int[W + 1];

        for ( i = 0; i < n; i++)
            for ( j = 0; j < W + 1; j++)
                dp[i][j] = -1;
        return knapSackRec(W, wt, val, n - 1, dp);
```

Complexity Analysis:
Time Complexity: O(N*W).
As redundant calculations of states are avoided.



Auxiliary Space: O(N*W).
The use of 2D array data structure for storing intermediate states

The most optimal solution to the problem will be dp[N][W] i.e. max value that can be obtained upto index N with max capacity of W

Code (Bottom up Approach) :

```
function main()
    val[] = { 60, 100, 120 };
    wt[] = { 10, 20, 30 };
    W = 50;
    n = sizeof(val) / sizeof(val[0]);
    print knapSack(W, wt, val, n);

function max(int a, int b)
    return(a > b) ? a : b;
function knapSack(int W, int wt[], int val[], int
n)
    i, w;
    K[n + 1][W + 1];
    for(i = 0; i <= n; i++)
        for(w = 0; w <= W; w++)
            if (i == 0 || w == 0)
    then
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
    then
                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];
        endfor
    endfor
    returnK[n][W];
```

Illustration :
Let weight elements = {1, 2, 3}
Let weight values = {10, 15, 40}
Capacity=6

	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						

Explanation:
For filling 'weight = 2' we come
across 'j = 3' in which
we take maximum of
(10, 15 + DP[1][3-2]) = 25
  | |
'2'   '2 filled'
not filled

	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

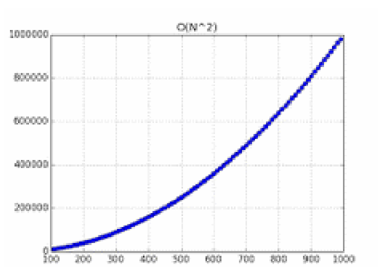
Explanation:
For filling 'weight=3',
we come across 'j=4' in which
we take maximum of (25, 40 + DP[2][4-3])
= 50

For filling 'weight=3'
we come across 'j=5' in which
we take maximum of (25, 40 + DP[2][5-3])
= 55

For filling 'weight=3'
we come across 'j=6' in which
we take maximum of (25, 40 + DP[2][6-3])
= 65

Complexity Analysis:

- **Time Complexity:** O(N*W).
where 'N' is the number of weight elements and 'W' is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W



- **Auxiliary Space:** O(N*W).
The use of 2-D array of size 'N*W'. **Can we do better?**

  If we observe carefully, we can see that the dp solution with states (i,j) will depend on state

  (i-1, j) or (i-1, j-wt[i-1]). In either case the solution for state (i,j) will lie in the i-1th row of the memoization table. So at every iteration of the index, we can copy the values of the current row and use only this row for building the solution in the next iteration and no other row will be used. Hence, at any iteration we will be using only a single row to build the solution for the current row. Hence, we can reduce the space complexity to just O(W).

Space-Optimized DP Code (for Bottom up approach) :

```
function main()
    val[] = {7, 8, 4}, wt[] = {3, 8, 6}, W = 10, n = 3;
    print KnapSack(val, wt, n, W) << endl;
    return 0;

function KnapSack(int val[], int wt[], int n, int W)
{
    int mat[2][W+1];
    memset(mat, 0, sizeof(mat));

    int i = 0;
    while (i < n)
        int j = 0;
        if (i%2!=0)
        then
            while (++j <= W)
                if (wt[i] <= j)
                then
                    mat[1][j] = max(val[i] + mat[0][j-wt[i]],
                                    mat[0][j] );
                else
                    mat[1][j] = mat[0][j];
            endwhile
        endif

        else
            while(++j <= W)
                if (wt[i] <= j)
```

then

mat[0][j] = **max**(val[i] + mat[1][j-wt[i]],

mat[1][j]);

else

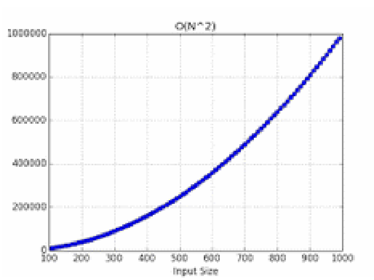
mat[0][j] = mat[1][j];

i++;

return(n%2 != 0)? mat[0][W] : mat[1][W];

Complexity Analysis:

- **Time Complexity**: O(N*W)



- **Space Complexity**: O(W)

IV. Algorithm Analysis

We have 3 different approaches to solve this question of 0/1 Knapsack. First one is the basic brute force approach in which we apply recursion. Space complexity of this approach is O(n)

Where n is the number of items in the list and the time complexity of this approach is exponential i.e.,O(2ˆn). This solution is easy to think about but very less efficient due to its execution time.

Next approach is the basic optimization over the basic recursive approach . We store answers for each recursive call each time any recursive call is called for the first time.Later we directly use the stored value rather than calling again.

Time complexity for this approach is O(n*m) where n is the number of items in the list and m is the capacity of the knapsack. space complexity will be O(n*m) due to this 2D array for storing results for the recursive calls .

Last approach is bottom up approach. This algorithm's time complexity is O(n*m)

V. Conclusion

We can conclude that both the dynamic programming solutions are really efficient .

But now let us discuss whether this solution will work in parallel.

A **parallel algorithm** is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result. We have used DP which divides problems into subproblems but this must be noticed that subproblems are not independent . And we know that dynamic programming can't be solved in parallel because each **subproblem** depends on the result given by other subproblems .

So we reached the conclusion that a parallel algorithm is not a good idea for a 0/1 knapsack problem.

VI. References

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp

https://www.geeksforgeeks.org/space-optimized-dp-solu
?ref=rp

https://www.educative.io/blog/0-1-knapsack-problem-dy

https://www.educative.io/courses/
grokking-dynamic-programming-patterns-for-coding-inte
RM1BDv71V60