0/1 Knapsack Problem
(using BRANCH & BOUND)

Presented by

41.ABHISHEK KUMAR SINGH

0/1 Knapsack Problem

Given two integer arrays val[0..n-1] and wt[0..n-1] that represent values and weights associated with n items respectively.
Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W.

We have 'n' items with value v_1 , v_2 . . . v_n and weight of the corresponding items is w_1 , w_2 . . . W_n .
Max capacity is W .

We can either choose or not choose an item. We have x_1 , x_2 . . . x_n.
Here x_i = { 1 , 0 }.

x_i = 1 , item chosen
x_i = 0 , item not chosen

Different approaches of this problem :

- **Dynamic programming**
- **Brute force**
- **Backtracking**
- **Branch and bound**

Why branch and bound ?

- **Greedy approach** works only for fractional knapsack problem.
- If weights are not integers , **dynamic programming** will not work.
- There are 2^n possible combinations of item , complexity for **brute force** goes exponentially.

What is branch and bound ?

- **Branch and bound is an algorithm design paradigm which is generally used for solving *combinatorial optimization problems.***
- **These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.**
- **Branch and Bound solve these problems relatively quickly.**

- ***Combinatorial optimization problem*** is an optimization problem, where an optimal solution has to be identified from a finite set of solutions.

ALGORITHM

- **Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.**
- **Initialize maximum profit, maxProfit = 0**
- **Create an empty queue, Q.**
- **Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.**
-

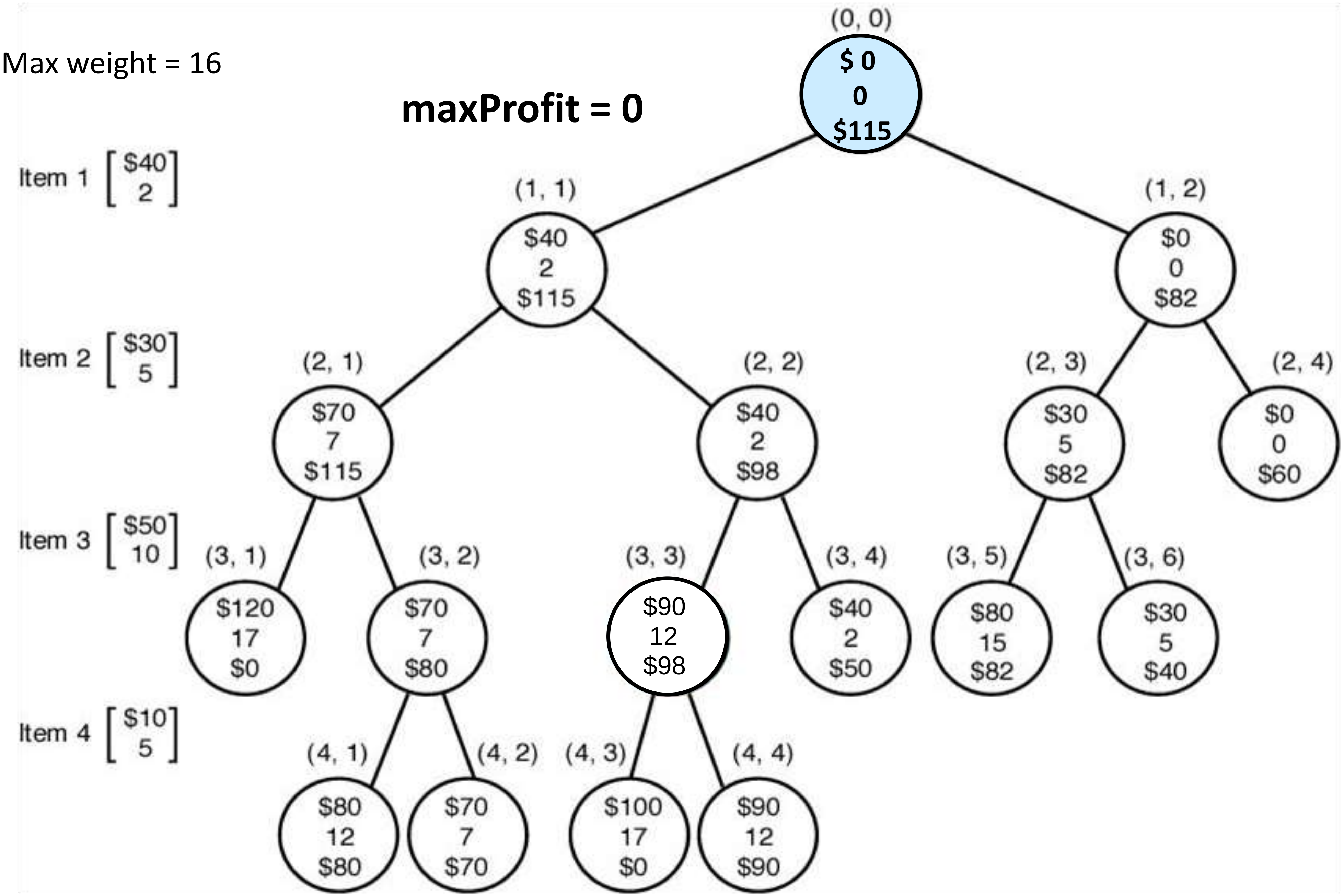
- Do following while Q is not empty.
-
  - Extract an item from Q. Let the extracted item be u.
  - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
  - Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
  - Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

➢

EXAMPLE :

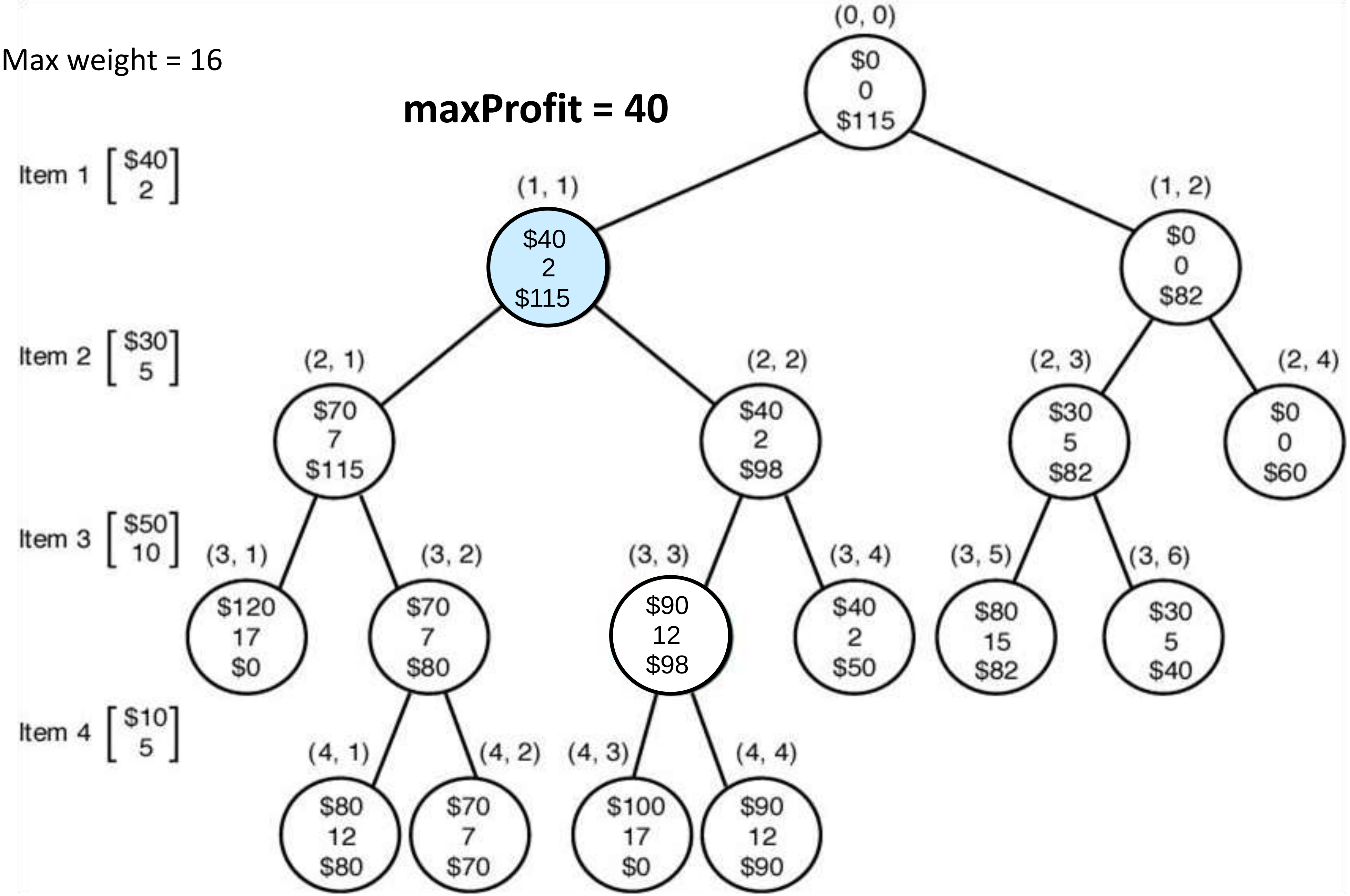
Max weight = 16

maxProfit = 0



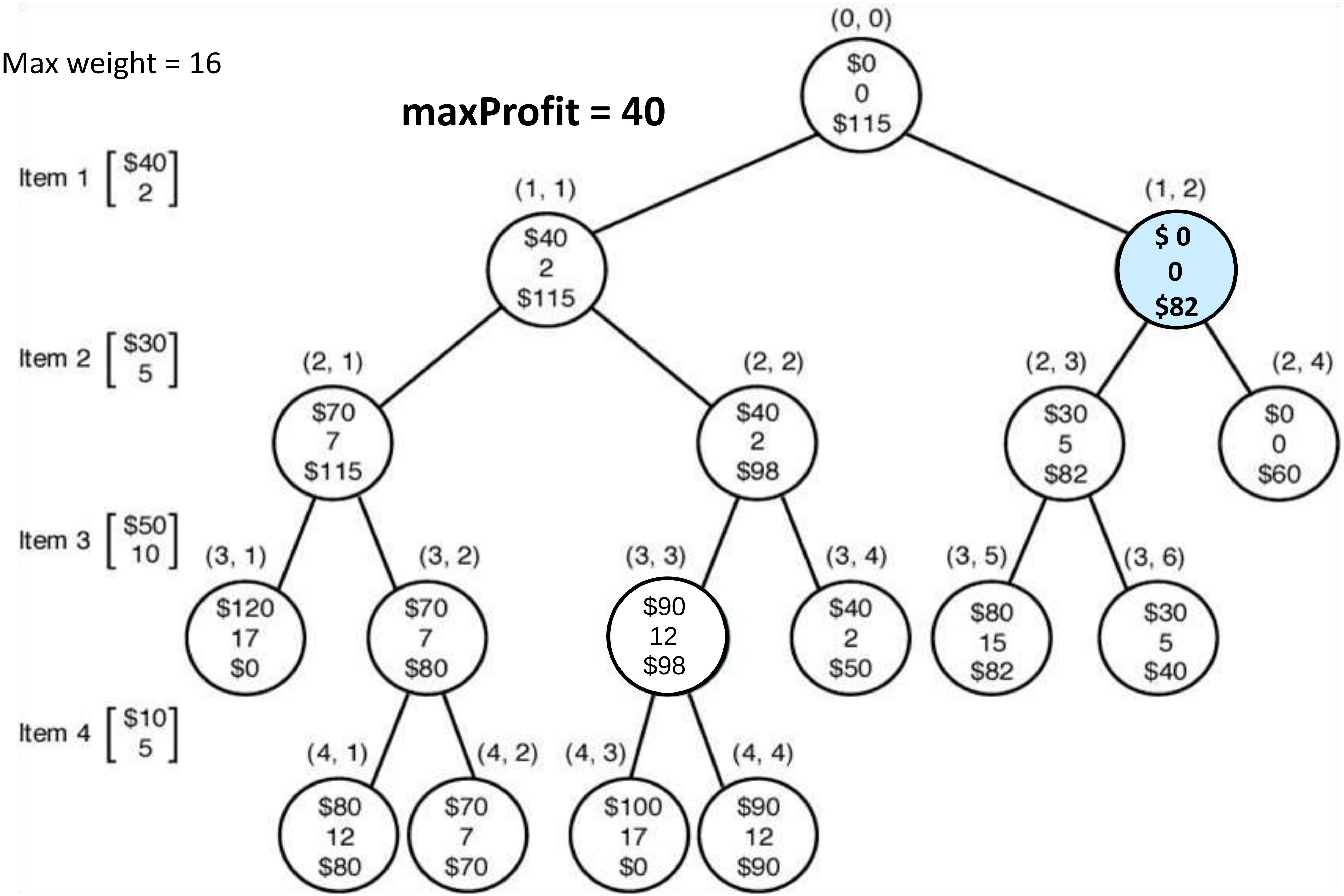
Max weight = 16

maxProfit = 40



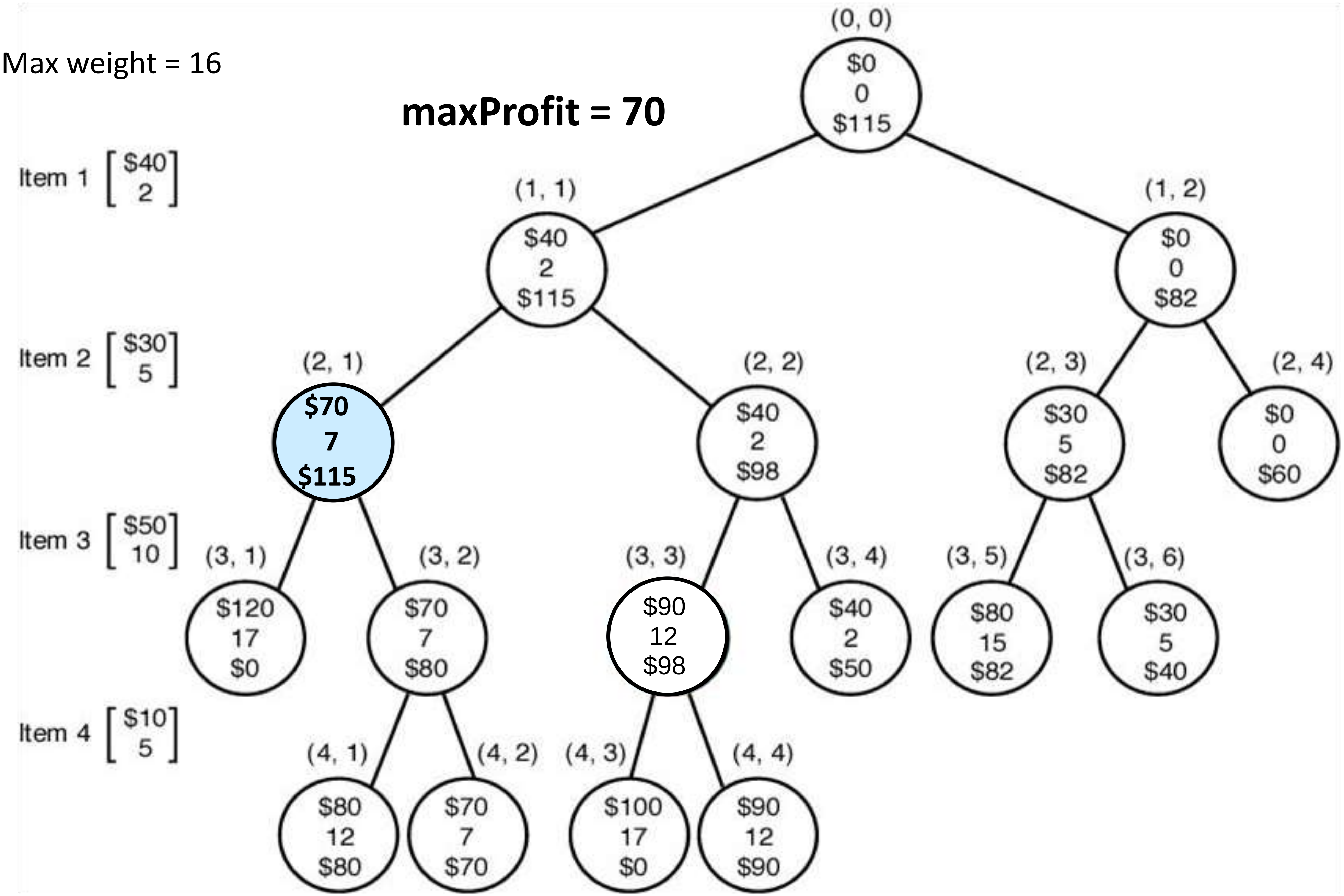
Max weight = 16

maxProfit = 40



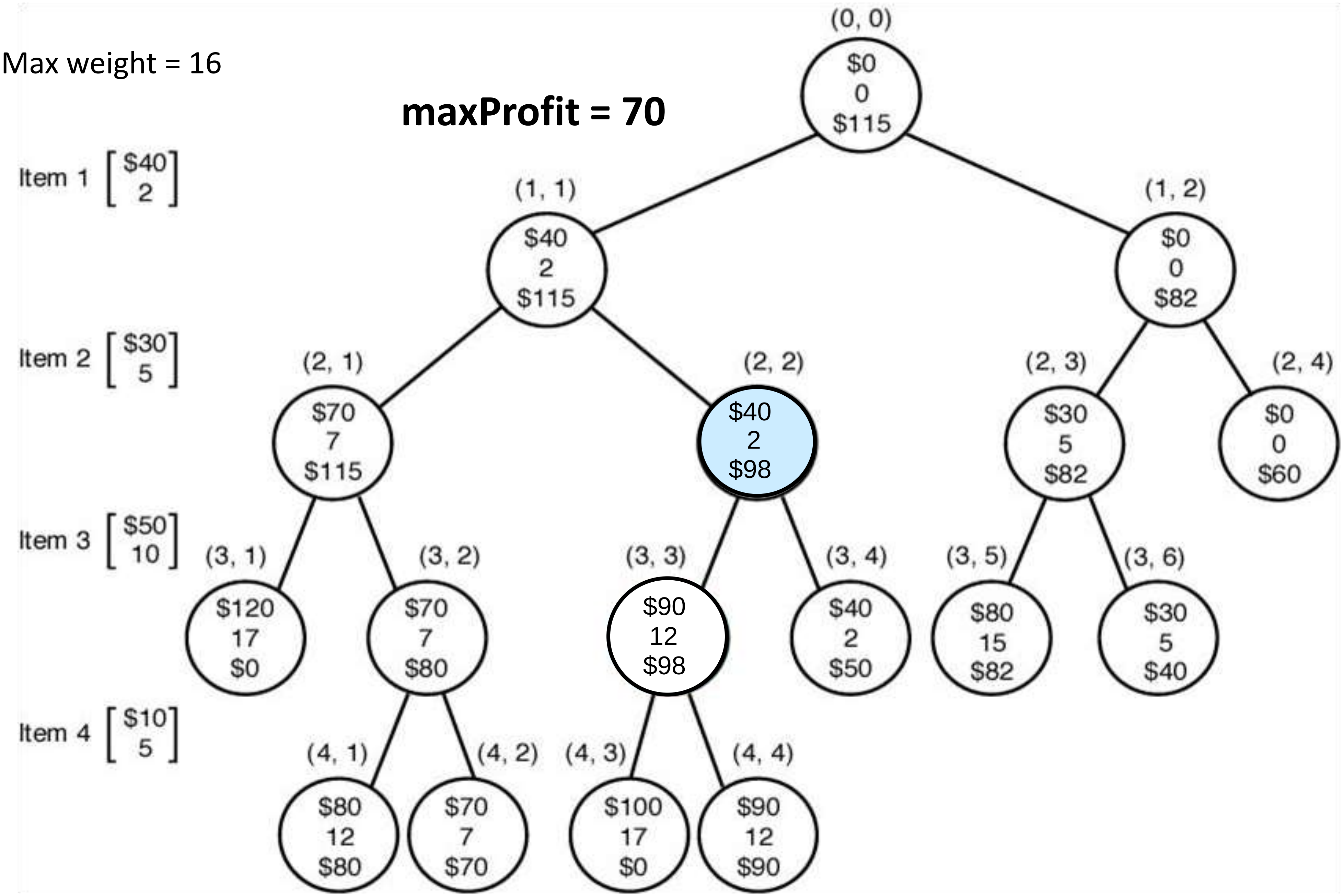
Max weight = 16

maxProfit = 70



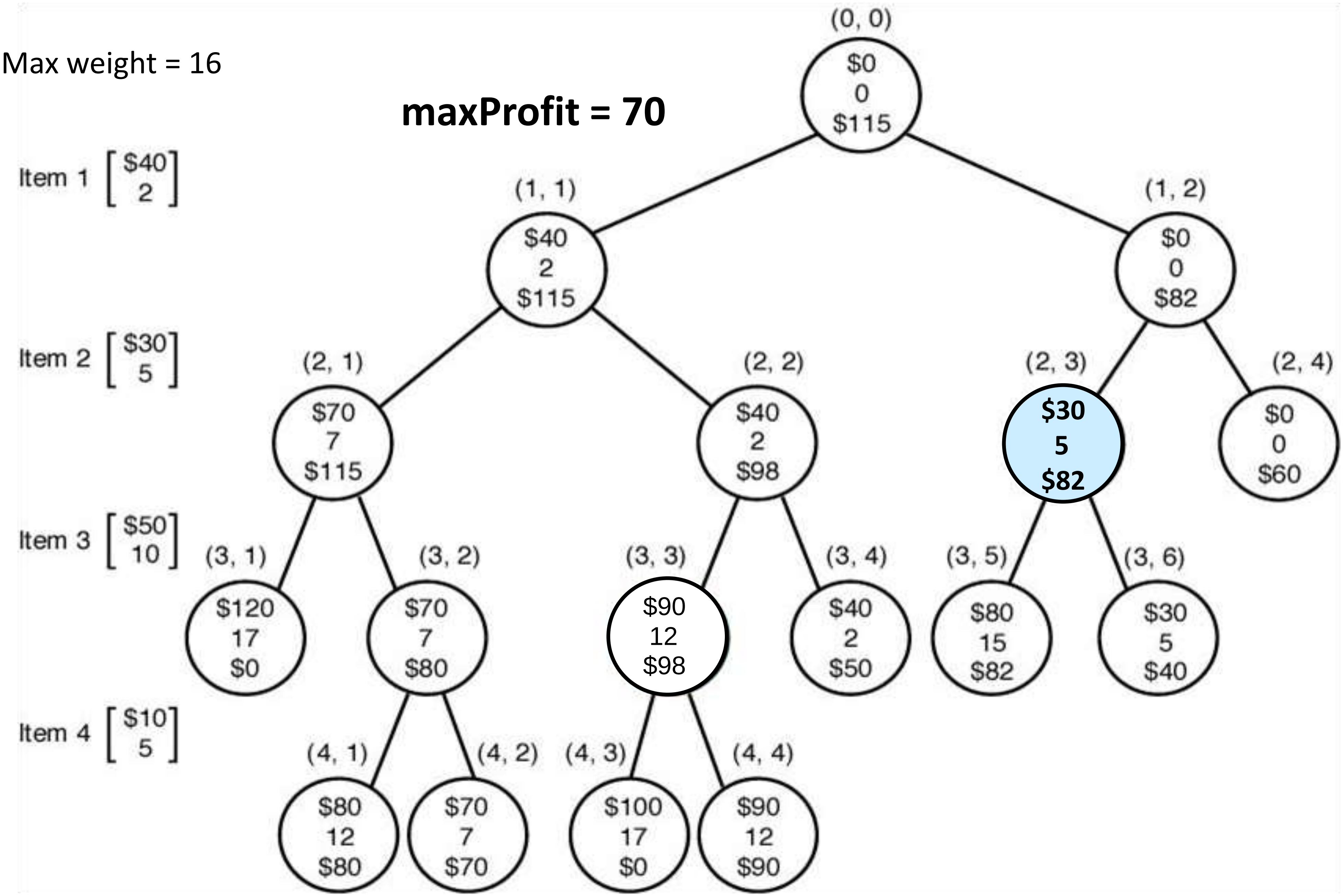
Max weight = 16

maxProfit = 70



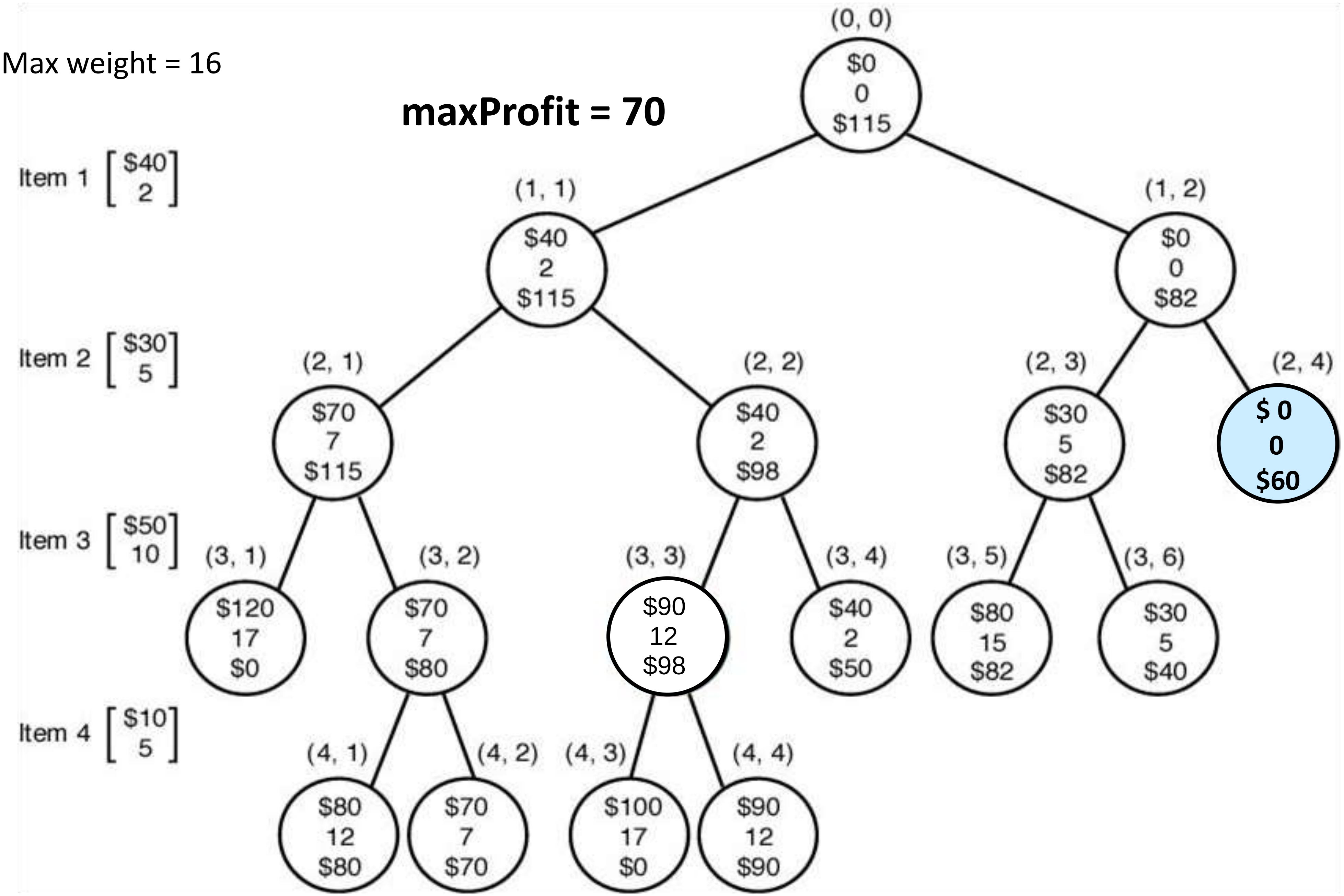
Max weight = 16

maxProfit = 70



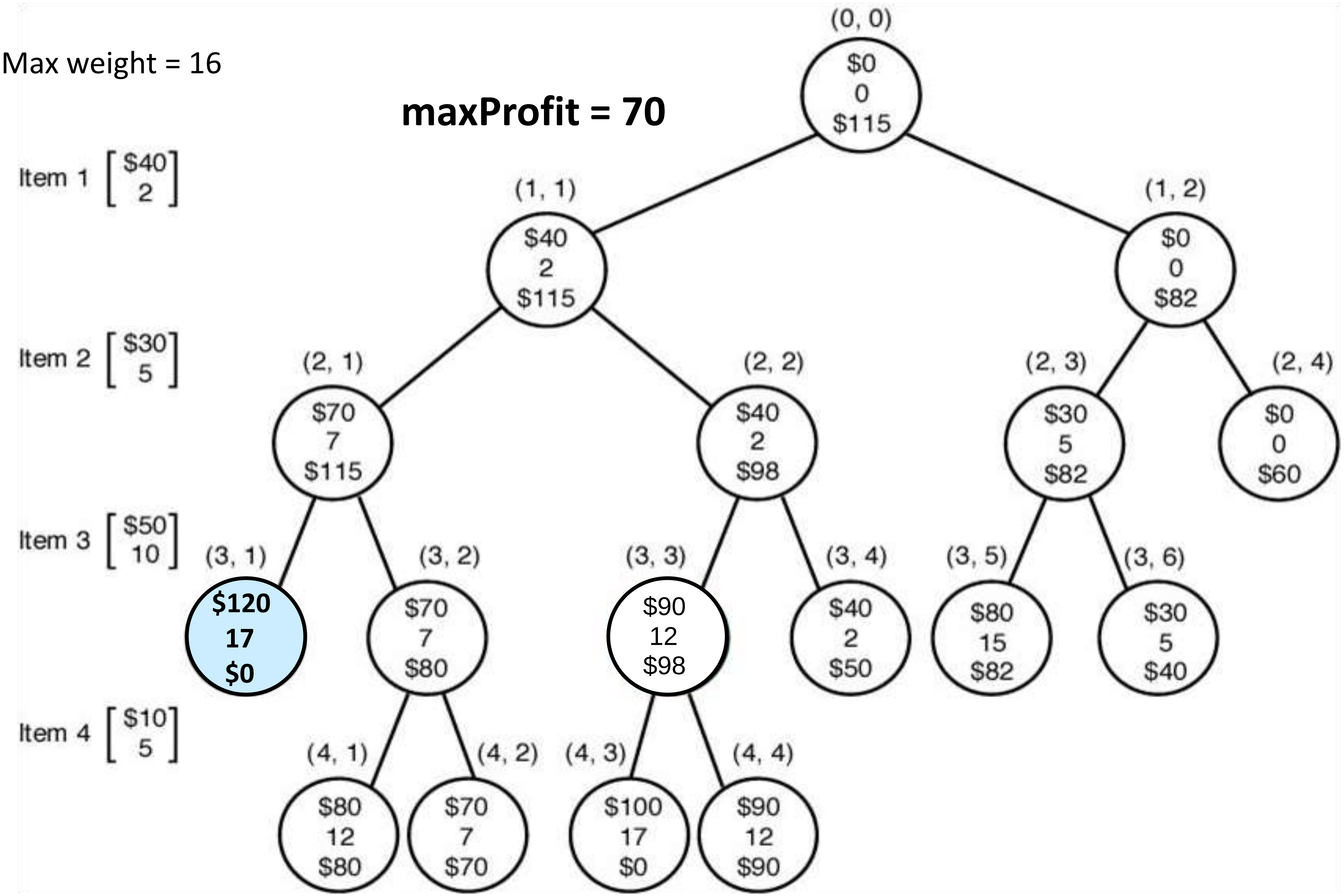
Max weight = 16

maxProfit = 70



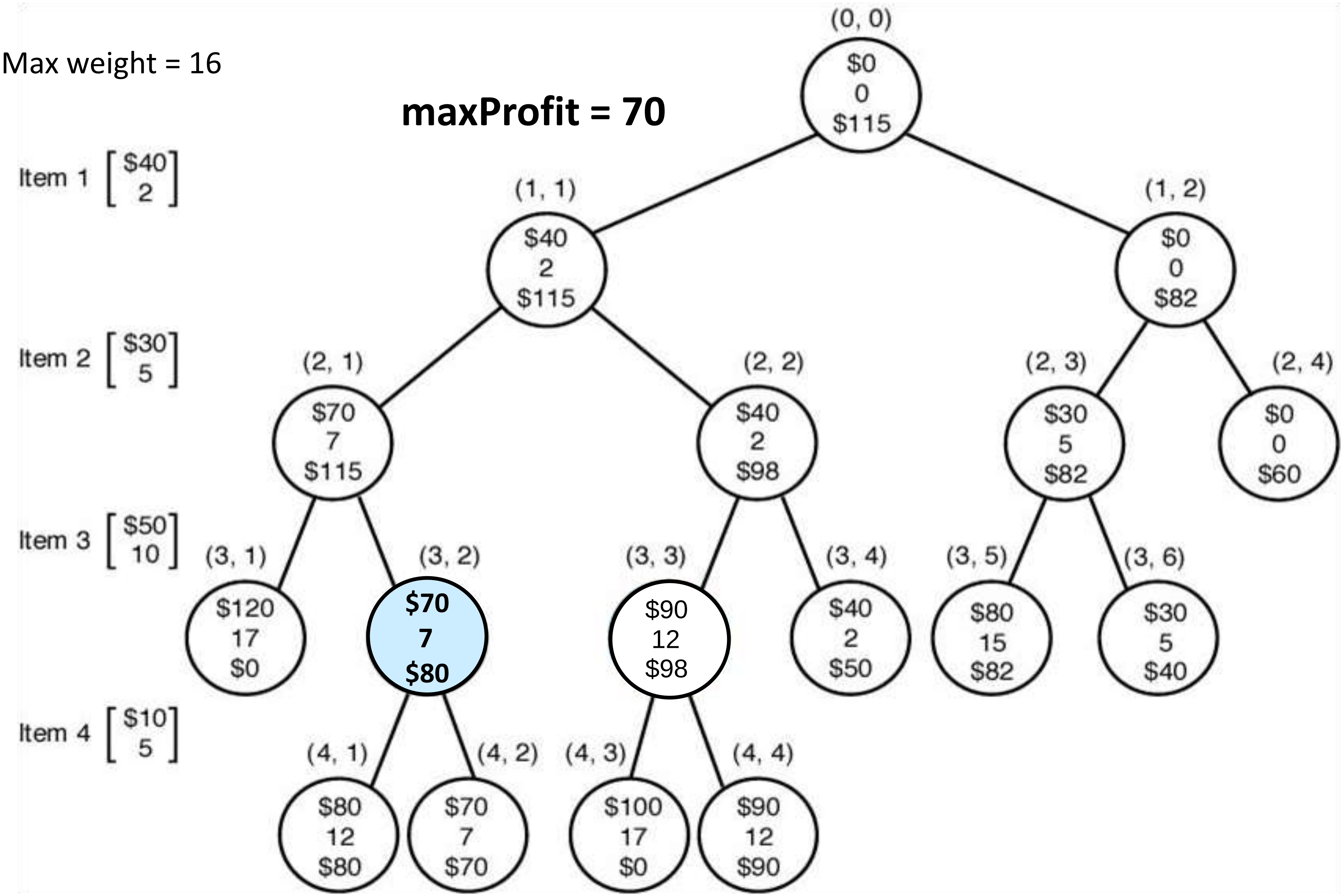
Max weight = 16

maxProfit = 70



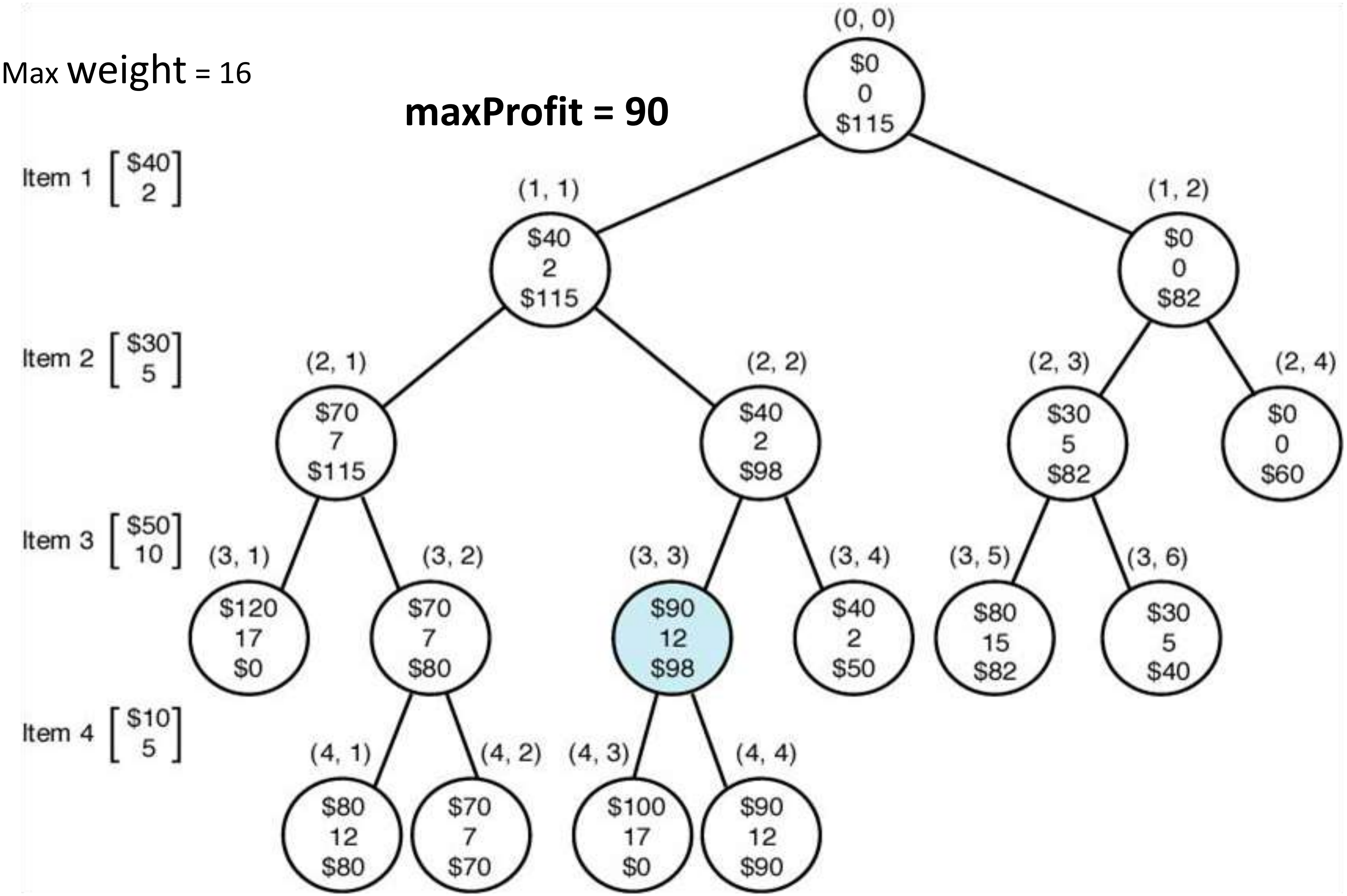
Max weight = 16

maxProfit = 70



Max weight = 16

maxProfit = 90



Data items used in the Algorithm :

```
struct Item
{
        float weight;
        int value;
}
```

Node structure to store information of decision tree

```
struct Node
{
        int level, profit, bound;
        float weight;
   // level     ---> Level of node in decision tree (or index ) in arr[]
   // profit   ---> Profit of nodes on path from root to this node (including this node)
   // bound ---> Upper bound of maximum profit in subtree of this node
}
```

Algorithm for maxProfit :

```
knapsack(int W, Item arr[], int n)
        queue<Node> Q
        Node u, v                              //u.level = -1
        Q.push(u)                              //u.profit = u.weight = 0
        while ( !Q.empty() )                   //int maxProfit = 0
                u = Q.front()  &  Q.pop()
                v.level = u.level + 1              //  selecting the item
                v.weight = u.weight + arr[v.level].weight
                v.profit = u.profit + arr[v.level].value
                if (v.weight <= W && v.profit > maxProfit)
                        maxProfit = v.profit
                v.bound = bound(v, n, W, arr)
                if (v.bound > maxProfit)
                        Q.push(v)
                v.weight = u.weight                // not selecting the item
                v.profit = u.profit
                v.bound = bound(v, n, W, arr)
                If (v.bound > maxProfit)
                        Q.push(v)
        return (maxProfit)
```

Procedure to calculate upper bound :

```
bound(Node u, int n, int W, Item a[])
    if (u.weight >= W)
        return (0)
    int u_bound <- u.profit
    int j <- u.level + 1
    int totweight <- u.weight
    while ((j < n) && (totweight + a[j].weight <= W))
        totweight <- totweight + a[j].weight
        u_bound <- u_bound + a[j].value
        j++
    if (j < n)
        u_bound <- u_bound + ( W - totweight ) * a[j].value /a[j].weight
    return (u_bound)
```

THANK YOU