Solving 0/1 Knapsack problem using Recursion

Posted on October 8, 2016 by dharam

Introduction

A knapsack is a bag with straps, usually carried by soldiers to help them take their valuables or things which they might need during their journey. The 0/1 knapsack problem is a very famous interview problem. The problem statement is as follows:

Given a set of items, each of which is associated with some weight and value. Find the subset of items which can be carried in a knapsack of capacity W (where W is the weight). It is required that the cumulative value of the items in the knapsack is maximum value possible.

In simple words, it asks you to pick certain items from the set of items such that their total weight is less than or equal to W and the sum of their values is maximum.

What is the meaning of 0/1?

0/1 means that either we can pick an item or we can leave the item. It is impossible to take a fraction of the item.

Initial thought process

An initial brute force solution to this problem is to filter all combinations of items such that their total weights is less than or equal to W. Then calculate values of all such combinations and return the one which has the maximum value.

This works well, but the total number of possible combinations would be 2^N if there are N items. We arrive at this number because each item can either be picked or left. However, some optimizations can be done to discard items with individual weights more than W and then find all combinations.

But still this is a huge number, exponential to be more precise.

How to put this into a solution

Here is a simple idea, let us say that we have an array of items. For the first item, we have two possibilities, either pick it or leave it. Hence, we are talking about two different branches of the solution. Similarly for these two branches, we have two options for the second item and so on.

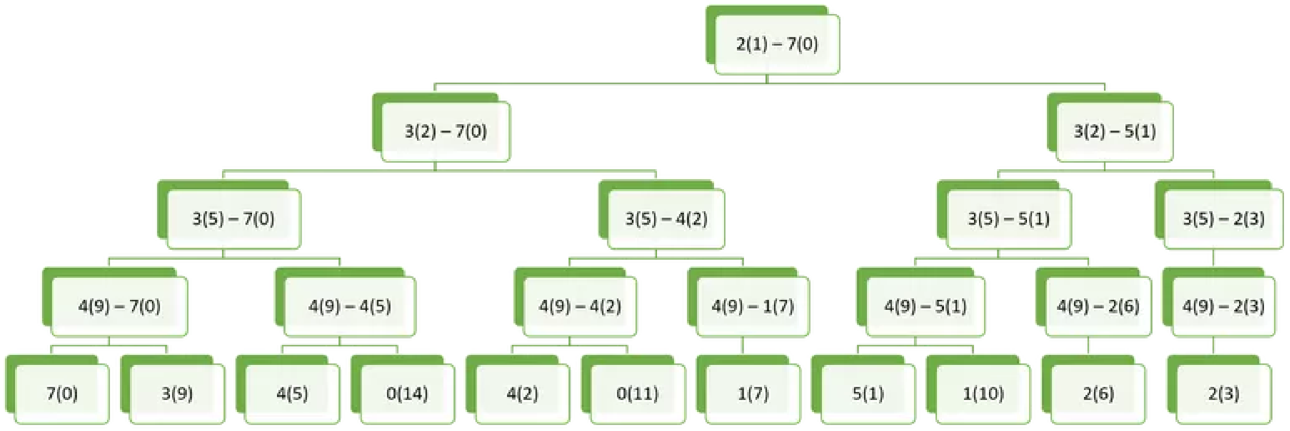
For e.g. Below is the set of items with weights and values

Weights : 2 3 3 4

Values :   1 2 5 9

Knapsack Capacity (W) = 7

Here is a pictorial representation of the idea



Each level represents a new item. Let me first tell you what is the value in each node. The data format is X(Y) – A(B). This means, the item has weight X and value Y. Also, at this level A is the capacity remaining and B is the value we have achieved till now.

Now, the root node 2(1) – 7(0) means, the item under consideration has weight 2 and value 1. At this point, we still have remaining capacity 7 and the value accumulated is 0. This is correct because this is the first item and we haven't chosen any item till now. Similarly, the right child of root represents, the item with weight 3 and value 2 and at this point the capacity remaining is 5 and value accumulated is 1.

So, why do we branch into left and right?

At each node we branch into left and right, the left branch is the solution when we do not choose the item at the parent node. The right branch is the solution when we have already picked up the item at the parent node.

What happens when we pick an item?

Once we pick an item, there are two effects:

- The remaining capacity of knapsack decreases by the weight of the chosen item
- The accumulated value increases by the value of the chosen item.

For e.g.

The right child of root means, we have picked the root, hence the remaining capacity is 5 and the accumulated value is 1. Where as the left child of the root means, we haven't picked the item, hence the remaining capacity is 7 and the value accumulated is 0.

Now as we all understand that we can only pick items till our remaining capacity is greater than or equal to the weight of the item. There for the right most leaf with value 3(5) – 2(3) wont have the right child.

The final answer to the question would be the leaf node which has maximum value, in our case it would be the node with value 0(14) .

Source Code

```java
private static int knapsack(int[] weight, int[] val, int w, int itemNum) {
  if (w == 0 || itemNum == weight.length) {
    return 0;
  }
  if (weight[itemNum] > w)
    return knapsack(weight, val, w, itemNum + 1);

  int rMax = val[itemNum] + knapsack(weight, val, w - weight[itemNum], itemNum + 1);
  int lMax = knapsack(weight, val, w, itemNum + 1);
  return Math.max(rMax, lMax);
}
```

The above code takes the weight and the value array, it also takes the remaining capacity and the item under consideration.

At each recursive step, if the item's individual weight is less that the remaining capacity, it tries to branch one after picking the item and the other without picking the item. Once it calculates the maximum value from both the paths, it return the max of both the values.

Analysis

Clearly, this code requires a recursion stack, hence the space complexity is equal to the depth of the stack. In our demonstration above, you can see that the recursion stack is N level deep where N is the number of items or the length of the item array. We can say that the space complexity is O(N).

The running time of this algorithm can be written as the following recurrence:

T(N) = 2T(N-1) + O(1), which is simplified to O(2^N). This is also evident from the recursion tree, which has 2^N leaves.

Readers interested in dynamic programming

Many readers ask me how to know if a problem can be solved using dynamic programming. Look at the above, you will find two types of behavior:

- Overlapping sub problems at the third level. Look at the nodes with values 4(9)-4 and 4(9) – 2. There are two instances of each. These instances are exactly same, hence if solved once, you might not want to solve it twice. A good candidate for memoization. If the tree was 10 – 15 levels deep, there would have been many more such instances. This is the first indication that a problem can be solved using dynamic programming
- Optimal Substructure, when we solved this problem, we were looking ofr maximum accumulated value at each level, in conjunction to other selected items. This means, that the smaller solution at each level, was a contributing factor to a bigger solution

Read my next post about solving the same problem with dynamic programming

Summary

The above solution is of course not efficient, you can introduce memoization to remember few of the repeating solutions, so that the work done for a similar node in the tree is constant time instead of exponential.

Please comment below if you need that code, I would try and add it here itself.

Related

Solving 0/1 Knapsack problem using Dynamic Programming
October 8, 2016
In "Algorithms"

Data Structures and Algorithms – Recursion
October 20, 2013
In "Algorithms"

Dynamic Programming - Longest Common Subsequence
August 10, 2013
In "Algorithms"

Posted in Algorithms, Recursion ? Tagged knapsack, recursion

Find Interleaving Strings using Dynamic Programming

Solving 0/1 Knapsack problem using Dynamic Programming