Dynamic Programming: Subset Sum & Knapsack

Slides by Carl Kingsford

Mar. 22, 2013

Based on AD Section 6.4

Dynamic Programming

Extremely general algorithm design technique

Similar to divide & conquer:

- Build up the answer from smaller subproblems
- More general than "simple" divide & conquer
- Also more powerful

Generally applies to algorithms where the brute force algorithm would be exponential.

Subset Sum

Problem (Subset Sum). *Given:*

- *an integer bound* W, *and*
- *a collection of* n *items, each with a positive, integer weight* w_i,

find a subset S *of items that:*

$$\text{maximizes } \sum_{i \in S} w_i \text{ while keeping } \sum_{i \in S} w_i \leq W.$$

Motivation: you have a CPU with W free cycles, and want to choose the set of jobs (each taking w_i time) that minimizes the number of idle cycles.

Assumption

$$\boxed{\text{We assume } W \text{ and each } w_i \text{ is an integer.}}$$

Just look for the value of the OPT

Suppose for now we're not interested in the actual set of intervals.

Only interested in the *value* of a solution
(aka it's cost, score, objective value).

This is typical of DP algorithms:

- You want to find a solution that optimizes some value.
- You first focus on just computing what that optimal value would be. E.g. what's the highest weight of a set of items?
- You then post-process your answer (and some tables you've created along the way) to get the actual solution.

Optimal Notation

Notation:

- Let S^* be an optimal choice of items (e.g. a set {1,4,8}).
- Let $OPT(n, W)$ be the value of the optimal solution.
- We design an dynamic programming algorithm to compute $OPT(n, W)$.

Subproblems:

- To compute $OPT(n, W)$: We need the optimal value for subproblems consisting of the first j items for every knapsack size $0 \leq w \leq W$.
- Denote the optimal value of these subproblems by $OPT(j, w)$.

Recurrence

Recurrence: How do we compute $OPT(j, w)$ in terms of solutions to smaller subproblems?

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$

$$\begin{aligned} OPT(0, W) &= 0 \qquad \text{If no items, 0} \\ OPT(j, 0) &= 0 \qquad \text{If no space, 0} \end{aligned}$$

Special case: if $w_j > W$ then $OPT(j, W) = OPT(j-1, W)$.

Another way to write it...

$$OPT(j, W) = \begin{cases} 0 & \text{if } j = 0 \text{ or } W = 0 \\ OPT(j-1, W) & \text{if } w_j > W \\ \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases} \end{cases}$$

The blue questions are different than the black questions: we don't know the answer to the black questions at the start.

So: we have to try both (that's what the max does).

The table of solutions

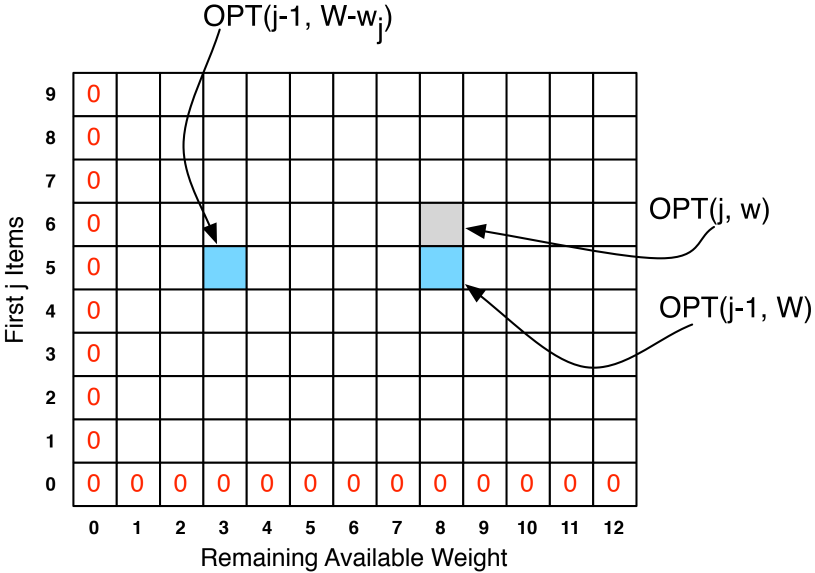
First j Items													
9	0												
8	0												
7	0												
6	0												
5	0												
4	0												
3	0												
2	0												
1	0												
0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12

Remaining Available Weight

OPT(n, W)

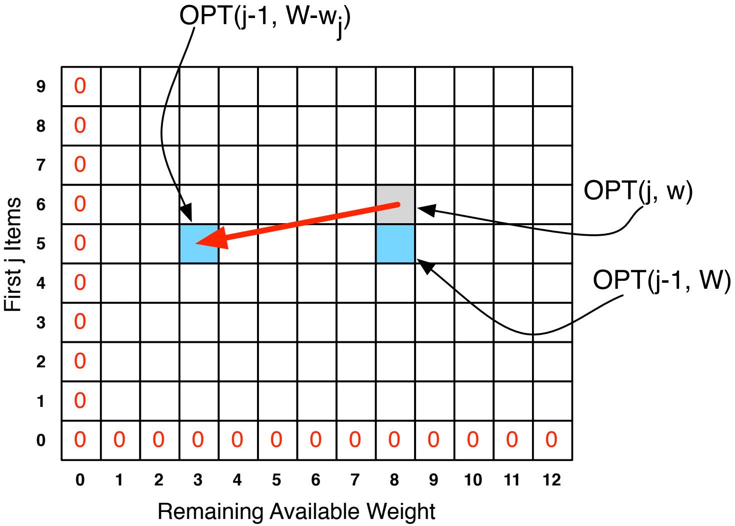
OPT(4, 11)

Filling in a box using smaller problems



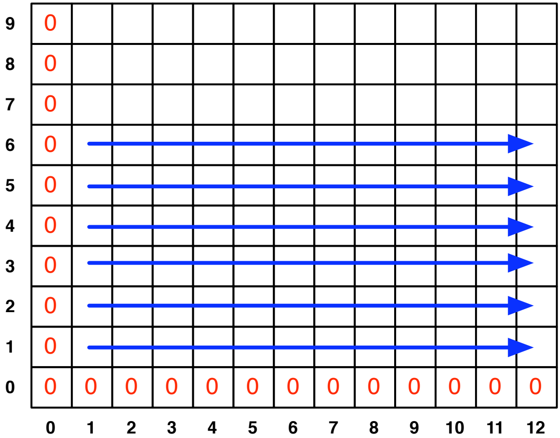
Remembering Which Subproblem Was Used

When we fill in the gray box, we also record which subproblem was chosen in the maximum:



Filling in the Matrix

Fill matrix from bottom to top, left to right.



When you are filling in box, you only need to look at boxes you've already filled in.

Pseudocode

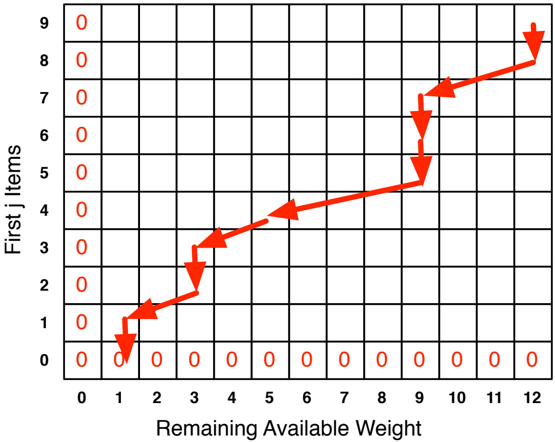
```
SubsetSum(n, W):
   Initialize M[0,r] = 0 for each r = 0,...,W
   Initialize M[j,0] = 0 for each j = 1,...,n

   For j = 1,...,n:                for every row
      For r = 0,...,W:             for every column
         If w[j] > r:              case where item can't fit
            M[j,r] = M[j-1,r]

         M[j,r] = max(             which is best?
            M[j-1,r],
            w[j] + M[j-1, W-w[j]]
         )
   Return M[n,W]
```

Finding The Choice of Items

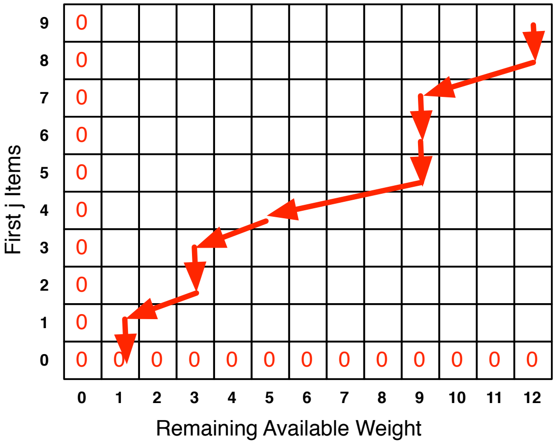
Follow the arrows backward starting at the top right:



Which items does this path imply?

Finding The Choice of Items

Follow the arrows backward starting at the top right:



Which items does this path imply? 8, 5, 4, 2

Runtime

- $O(nW)$ cells in the matrix.
- Each cell takes $O(1)$ time to fill.
- $O(n)$ time to follow the path backwards.
- Total running time is $O(nW + n) = O(nW)$.

This is pseudo-polynomial because it depends on the size of the input numbers.

General DP Principles

1. Optimal value of the original problem can be computed easily from some subproblems.

2. There are only a polynomial # of subproblems.

3. There is a “natural” ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.

General DP Principles

1. Optimal value of the original problem can be computed easily from some subproblems. $OPT(j, w) =$ max of two subproblems

2. There are only a polynomial # of subproblems. $\{(j, w)\}$ for $j = 1, \ldots, n$ and $w = 0, \ldots, W$

3. There is a "natural" ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems. Considering items $\{1, 2, 3\}$ is a smaller problem than considering items $\{1, 2, 3, 4\}$

Knapsack

Problem (Knapsack). *Given:*

- *a bound W, and*
- *a collection of n items, each with a weight w_i,*
- *a value v_i for each weight*

Find a subset S of items that:

$$\text{maximizes } \sum_{i \in S} v_i \text{ while keeping } \sum_{i \in S} w_i \leq W.$$

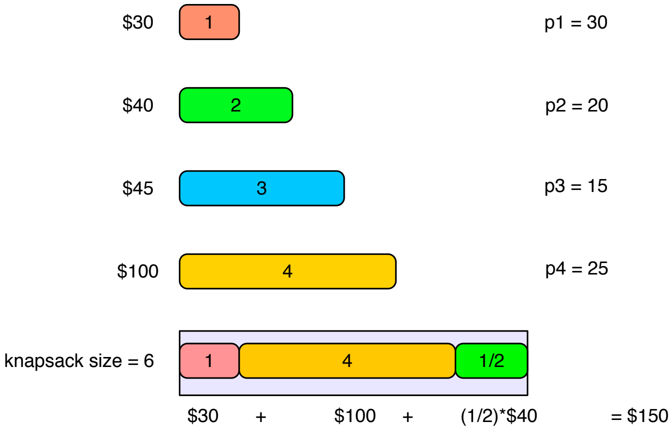
Difference from Subset Sum: want to maximize value instead of weight.

Why Greedy Doesn't work for Knapsack Example

Idea: Sort the items by $p_i = v_i / w_i$

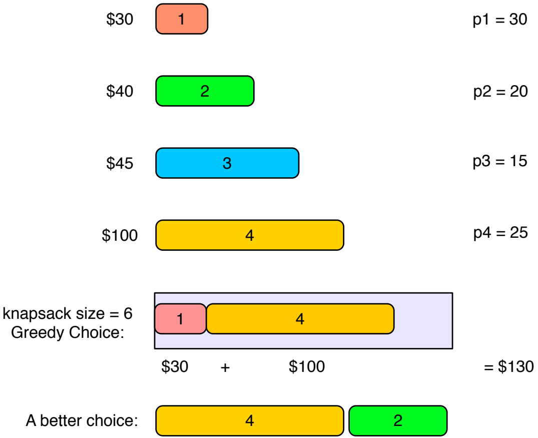
Larger v_i is better, smaller w_i is better.

If you were allowed to chose fractions of items, this would work:



0-1 Knapsack

This greedy algorithm doesn't work for knapsack where we can't take part of an item:



How can we solve Knapsack?

Knapsack

Subset Sum:

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$

Knapsack

Subset Sum:

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$

Knapsack:

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ v_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$