

Lecture 11: Dynamic Programming

CLRS Chapter 15

Outline of this section

- Introduction to **Dynamic programming**; a method for solving optimization problems.
- Dynamic programming vs. Divide and Conquer
- A few examples of Dynamic programming
 - the 0-1 Knapsack Problem
 - Chain Matrix Multiplication
 - All Pairs Shortest Path
 - The Floyd Warshall Algorithm: Improved All Pairs Shortest Path

Recalling Divide-and-Conquer

1. **Partition** the problem into particular subproblems.
2. **Solve** the subproblems.
3. **Combine** the solutions to solve the original one.

Remark: In the examples we saw the subproblems were usually **independent**, i.e. they did not call the same subsubproblems. If the subsubproblems were *not* independent, then D&C could be resolving many of the same problems many times. Thus, it does **more work than necessary!**

Dynamic programming (DP) solves every subsubproblem exactly once, and is therefore more efficient in those cases where the subsubproblems are not independent.

The Intuition behind Dynamic Programming

Dynamic programming is a method for solving optimization problems.

The idea: Compute the solutions to the subsub-problems *once* and store the solutions in a table, so that they can be *reused* (repeatedly) later.

Remark: We trade space for time.

0-1 Knapsack Problem

Informal Description: We have n items. Let v_i denote the value of the i -th item, and let w_i denote the weight of the i -th item. Suppose you are given a knapsack capable of holding total weight W .

Our goal is to use the knapsack to carry items, such that the total values are maximum; we want to find a subset of items to carry such that

- The total weight is at most W .
- The total value of the items is as large as possible.

We cannot take parts of items, it is the whole item or nothing. (This is why it is called **0-1**.)

How should we select the items?

0-1 Knapsack Problem

Formal description:

Given $W > 0$, and two n -tuples of positive numbers

$$\langle v_1, v_2, \dots, v_n \rangle \quad \text{and} \quad \langle w_1, w_2, \dots, w_n \rangle,$$

we wish to determine the subset

$T \subseteq \{1, 2, \dots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i,$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Remark: This is an optimization problem. The *Brute Force* solution is to try all 2^n possible subsets T .

Question: Is there a better way?

Yes. Dynamic Programming!

General Schema of a DP Solution

Step1: Structure: Characterize the structure of an optimal solution by showing that it can be decomposed into *optimal* subproblems

Step2: Recursively define the value of an optimal solution by expressing it in terms of optimal solutions for smaller problems (usually using min and/or max).

Step 3: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution: Construct an optimal solution from computed information.

Remarks on the Dynamic Programming Approach

- Steps 1-3 form the basis of a dynamic-programming solution to a problem.
- Step 4 can be omitted if only the value of an optimal solution is required.

Developing a DP Algorithm for Knapsack

Step 1: Decompose the problem into smaller problems.

We construct an array $V[0..n, 0..W]$.

For $1 \leq i \leq n$, and $0 \leq w \leq W$, the entry $V[i, w]$ will store the maximum (combined) value of any subset of items $\{1, 2, \dots, i\}$ of (combined) weight at most w .

That is

$$V[i, w] = \max \left\{ \sum_{j \in T} v_j : T \subseteq \{1, 2, \dots, i\}, \sum_{j \in T} w_j \leq w \right\}.$$

If we can compute all the entries of this array, then the array entry $V[n, W]$ will contain the solution to our problem.

Note: In what follows we will say that T is a *solution* for $[i, w]$ if $T \subseteq \{1, 2, \dots, i\}$ and $\sum_{j \in T} w_j \leq w$ and that T is an *optimal solution* for $[i, w]$ if T is a solution and $\sum_{j \in T} v_j = V[i, w]$.

Developing a DP Algorithm for Knapsack

Step 2: Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial Settings: Set

$$\begin{array}{ll} V[0, w] = 0 & \text{for } 0 \leq w \leq W, \quad \text{no item} \\ V[i, w] = -\infty & \text{for } w < 0, \quad \text{illegal} \end{array}$$

Recursive Step: Use

$$\begin{array}{l} V[i, w] = \max(V[i-1, w], v_i + V[i-1, w - w_i]) \\ \text{for } 1 \leq i \leq n, 0 \leq w \leq W. \end{array}$$

Intuitively, an optimal solution would either choose item i or not choose item i .

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computation of $V[i, w]$
(using iteration, not recursion).


Bottom: $V[0, w] = 0$ for all $0 \leq w \leq W$.

Bottom-up computation: Computing the table using

$$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$$

row by row.

$V[i, w]$	$w=0$	1	2	3	W	
$i=0$	0	0	0	0	0	bottom
1								
2								
\vdots								
n								



up

Example of the Bottom-up computation

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Remarks:

- The final output is $V[4, 10] = 90$.
- The method described does not tell which subset gives the optimal solution. (It is $\{2, 4\}$ in this example).

The Dynamic Programming Algorithm

```
KnapSack( $v, w, n, W$ )
{
  for ( $w = 0$  to  $W$ )  $V[0, w] = 0$ ;
  for ( $i = 1$  to  $n$ )
    for ( $w = 0$  to  $W$ )
      if ( $w[i] \leq w$ )
         $V[i, w] = \max\{V[i - 1, w], v[i] + V[i - 1, w - w[i]]\}$ ;
      else
         $V[i, w] = V[i - 1, w]$ ;
  return  $V[n, W]$ ;
}
```

Time complexity: Clearly, $O(nW)$.

Constructing the Optimal Solution

- The algorithm for computing $V[i, w]$ described in the previous slide does not record which subset of items gives the optimal solution.
- To compute the actual subset, we can add an auxiliary boolean array $keep[i, w]$ which is 1 if we decide to take the i -th file in $V[i, w]$ and 0 otherwise.

Question: How do we use all the values $keep[i, w]$ to determine the subset T of files having the maximum computing time?

Constructing the Optimal Solution

Question: How do we use the values $keep[i, w]$ to determine the subset T of items having the maximum computing time?

If $keep[n, W]$ is 1, then $n \in T$. We can now repeat this argument for $keep[n - 1, W - w_n]$.

If $keep[n, W]$ is 0, then $n \notin T$ and we repeat the argument for $keep[n - 1, W]$.

Therefore, the following partial program will output the elements of T :

```
 $K = W;$ 
for ( $i = n$  downto 1)
    if ( $keep[i, K] == 1$ )
    {
        output  $i$ ;
         $K = K - w[i];$ 
    }
```

The Complete Algorithm for the Knapsack Problem

```
KnapSack( $v, w, n, W$ )
{
  for ( $w = 0$  to  $W$ )  $V[0, w] = 0$ ;
  for ( $i = 1$  to  $n$ )
    for ( $w = 0$  to  $W$ )
      if ( $(w[i] \leq w)$  and  $(v[i] + V[i - 1, w - w[i]] > V[i - 1, w])$ )
      {
         $V[i, w] = v[i] + V[i - 1, w - w[i]]$ ;
         $keep[i, w] = 1$ ;
      }
      else
      {
         $V[i, w] = V[i - 1, w]$ ;
         $keep[i, w] = 0$ ;
      }
    }
   $K = W$ ;
  for ( $i = n$  downto 1)
    if ( $keep[i, K] == 1$ )
    {
      output  $i$ ;
       $K = K - w[i]$ ;
    }
  return  $V[n, W]$ ;
}
```

Dynamic Programming vs. Divide-and-Conquer

The Dynamic Programming algorithm developed runs in $O(nW)$ time.

We started by deriving a recurrence relation for solving the problem

$$V[0, w] = 0$$

$$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$$

Question: why can't we simply write a top-down divide-and-conquer algorithm based on this recurrence?

Answer: we could, but it could run in time $\Theta(2^n)$ since it might have to recompute the same values many times.

Dynamic programming saves us from having to recompute previously calculated subsolutions!

Final Comment

Divide-and-Conquer works **Top-Down**.

Dynamic programming works **Bottom-Up**.