

# Dynamic Programming

Subset Sum & Knapsack

02-251

Based on Algorithm Design by Kleinberg and Tardos Section 6.4

# Subset Sum

## 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

We assume  $W$  and each  $w_i$  is an integer.

# 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  $OPT(j, w)$  given solutions to smaller subproblems?

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

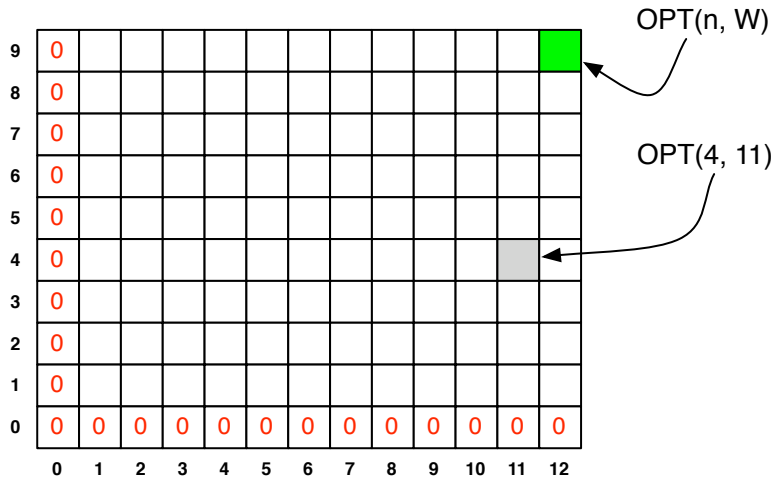
Special case: if  $w_j > W$  then  $OPT(j, W) = OPT(j-1, W)$ .

## Another way to write it...

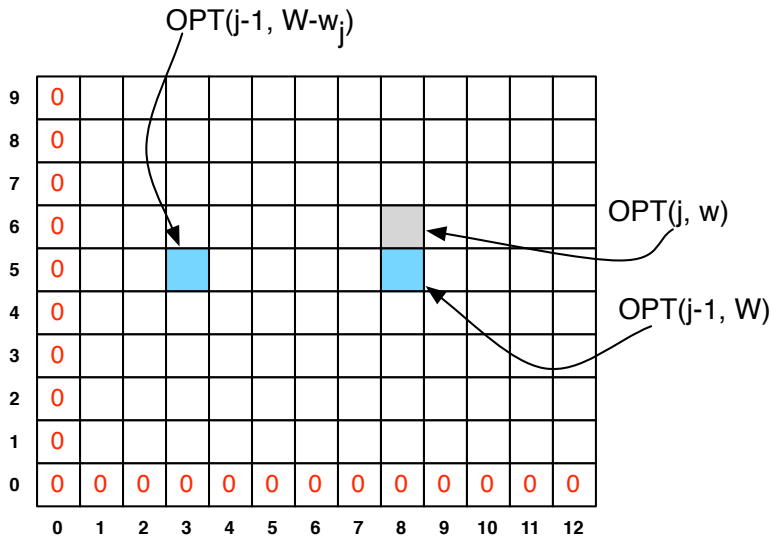
$$OPT(j, W) = \begin{cases} OPT(j-1, W) & \text{if } w_j > W \\ \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases} \end{cases}$$

Note: Because we don't know the answer to the blue questions, we have to try both.

## The table of solutions

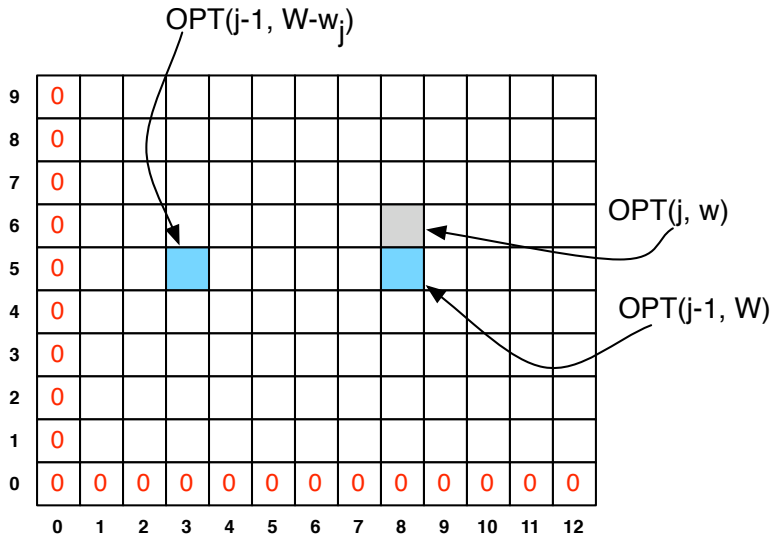


## Filling in a box using smaller problems



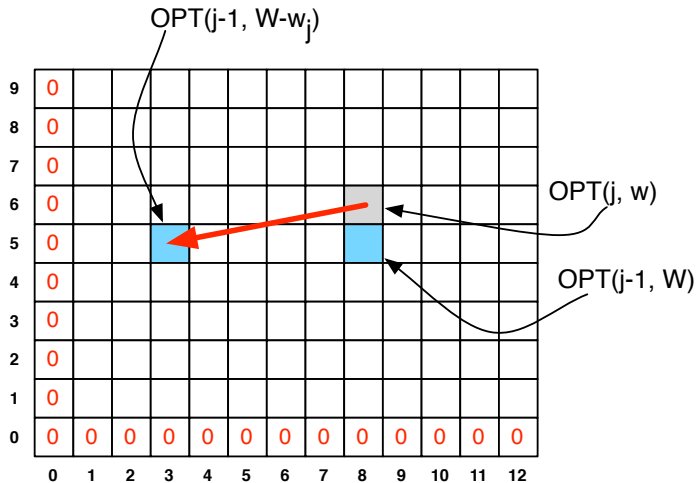


## Filling in a box using smaller problems



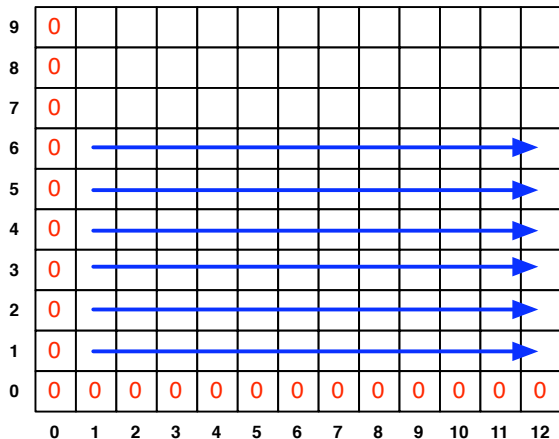
## Remembering Which Subproblem Was Used

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  $M[0,w] = 0$  for each  $w = 0, \dots, W$

Initialize  $M[i,0] = 0$  for each  $i = 1, \dots, n$

For  $i = 1, \dots, n$ :

for every row

For  $w = 0, \dots, W$ :

for every column

If  $w[i] > w$ :

case where item can't fit

$M[i,w] = M[i-1,w]$

$M[i,w] = \max($

which is best?

$M[i-1,w],$

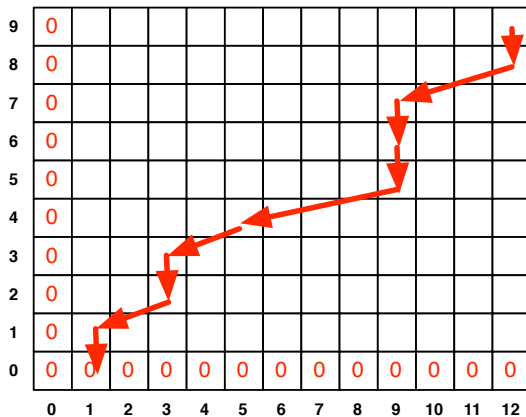
$w[j] + M[i-1, W-w[j]]$

)

Return  $M[n,W]$

# Finding The Choice of Items

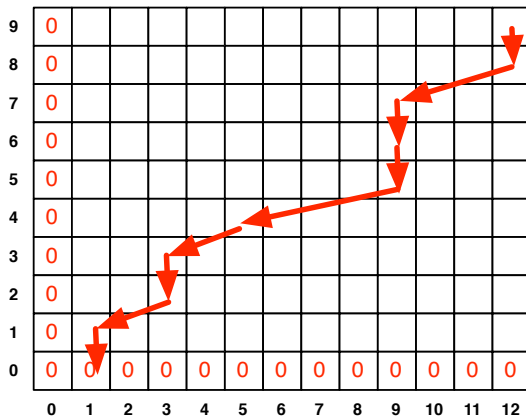
Follow the arrows backward starting at the top right:



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

## Runtime:

- $O(nW)$  to fill in the matrix.
- $O(n)$  time to follow the path backwards.
- Total running time is  $O(nW)$ .

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

# Knapsack

## 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:

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

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



How can we solve Knapsack?

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:

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

# Fractional Knapsack

## 0-1 Knapsack

You're presented with  $n$ , where item  $i$  has **value**  $v_i$  and **size**  $w_i$ . You have a knapsack of size  $W$ , and you want to take the items  $S$  so that

- $\sum_{i \in S} v_i$  is maximized, and
- $\sum_{i \in S} w_i \leq W$ .

This is a hard problem. However, if we are allowed to take **fractions** of items we can do it with a simple greedy algorithm:

- Value of a fraction  $f$  of item  $i$  is  $f \cdot v_i$
- Weight of a fraction  $f$  is  $f \cdot w_i$ .

# Knapsack Example

**Idea:** Sort the items by  $p_i = v_i/w_i$   
Larger  $v_i$  is better, smaller  $w_i$  is better.

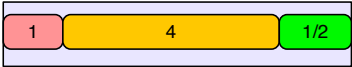
\$30      1       $p_1 = 30$

\$40      2       $p_2 = 20$

\$45      3       $p_3 = 15$

\$100      4       $p_4 = 25$

knapsack size = 6



The diagram shows a horizontal bar representing a knapsack. Inside the bar, from left to right, are: a red rounded rectangle labeled '1', a yellow rounded rectangle labeled '4', and a green rounded rectangle labeled '1/2'. The bar has a light purple background and a black border.

\$30    +    \$100    +     $(1/2)*\$40$     = \$150

# 0-1 Knapsack

This greedy algorithm doesn't work for 0-1 knapsack, where we can't take part of an item:

