Class 21

DP FOR INTEGER KNAPSACK

MEMOIZATION

Knapsack Complexity

n items; weights w_i; values v_i; capacity *W*

Dynamic Programming for Knapsack

Let $F(i, j)$ be the value of an optimal solution for the first i items that fit into a knapsack of capacity j.

This solution may not include item i.

In this case, the solution reduces to $F(i - 1, j)$

If item i is included in the solution then the solution consists of any subset that has capacity $j - w_i >= 0$.

The value of that subset is $v_i + F(i - 1, j - w_i)$

Dynamic Programming for Knapsack

We get:

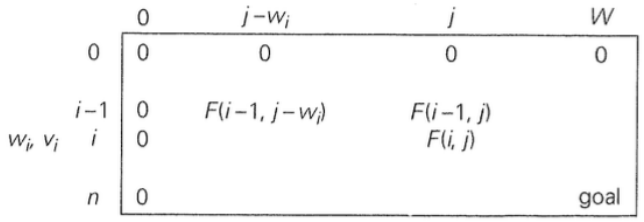
$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as:

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

Dynamic Programming for Knapsack

Graphically speaking, we get:



Dynamic Programming for Knapsack

Consider the following problem:

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

DP for Knapsack

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

We will solve it as follows:

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

DP with Memory for Knapsack

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                     Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

DP with Memory for Knapsack

```
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

DP for Knapsack Analysis

Table has nW entries

To determine each entry, 2 options. 1 subproblem for each.

Efficiency: $\Theta(nW)$

Can improve slightly if we work top-down recursively, and only compute table values once (we say the function computations are memoized). Will only compute table entries necessary to solve the overall problem

- Faster, but in most cases, stays $\Theta(nW)$
- If $\gcd(w_1,\ldots,w_n) = k$, then becomes $\Theta(nW/k)$