Objectives

- Dynamic Programming
  - Subset Sums
  - Knapsacks

The Price is Right

Or, shopping with someone else's money

- **Goal**: Spend as much money as possible without going over $100
  - CD $18
  - Jeans $40
  - DVD $35
  - Dinner $15
  - Book $8
  - Ice cream $5
  - Shoes $61
  - Pizza $7

Possible solutions?

Knapsack Problem

- Given *n* objects and a "knapsack"
- Item *i* weighs $w_i > 0$ kilograms and has value $v_i > 0$
  - Example: jobs require w_i time
- Knapsack has capacity of *W* kilograms
  - Example: *W* is time interval that resource is available

Goal: fill knapsack so as to maximize total value

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Towards a Recurrence…

- What do we know about the knapsack with respect to item *i*?

Towards a Recurrence…

- What do we know about the knapsack with respect to item *i*?
  - Either select item *i* or not
  - If don't select
    - Pick optimum solution of remaining items
  - Otherwise
    - What happens?
    - How does problem change?

Dynamic Programming: False Start

- Def. OPT(i) = max profit subset of items 1, …, i
  - Case 1: OPT does not select item i
    - OPT selects best of { 1, 2, …, i-1 }
  - Case 2: OPT selects item i
    - Accepting item *i* does not immediately imply that we will have to reject other items
      - No known conflicts
    - Without knowing what other items were selected before *i*, we don't even know if we have enough room for *i*

➡Need more sub-problems!

Dynamic Programming: Adding a New Variable

- **Def.** OPT(i, **w**) = max profit subset of items 1, …, i with weight limit **w**
  - Case 1: OPT does not select item *i*
    - OPT selects best of { 1, 2, …, i-1 } using weight limit ***w***
  - Case 2: OPT selects item *i*
    - new weight limit = $w - w_i$
    - OPT selects best of { 1, 2, …, i–1 } using new weight limit, $\boldsymbol{w - w_i}$

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

- Fill up an n-by-W array

```
Input: N, $w_1$,…,$w_N$, $v_1$,…,$v_N$

for w = 0 to W
   M[0, w] = 0

for i = 1 to N     # for all items
   for w = 1 to W  # for possible weights
      if $w_i$ > w :  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], $v_i$ + M[i-1, w-$w_i$] }

return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0											
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

OPT:
Value=

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

W + 1

N = 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

OPT:
Value=

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

W + 1

N = 2

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

OPT:
Value=

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

W + 1

N = 3

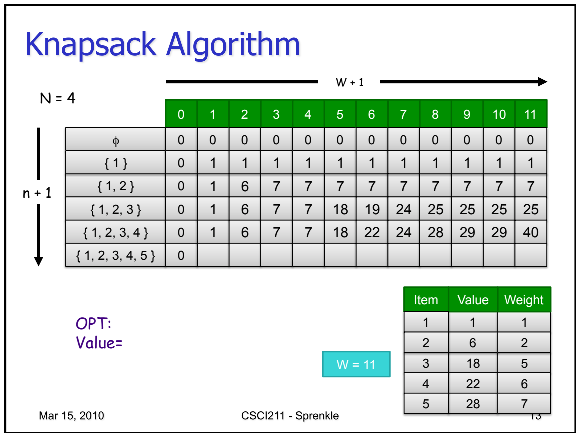
n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

OPT:
Value=

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7



Knapsack Algorithm

N = 4

W + 1 →

n + 1 ↓

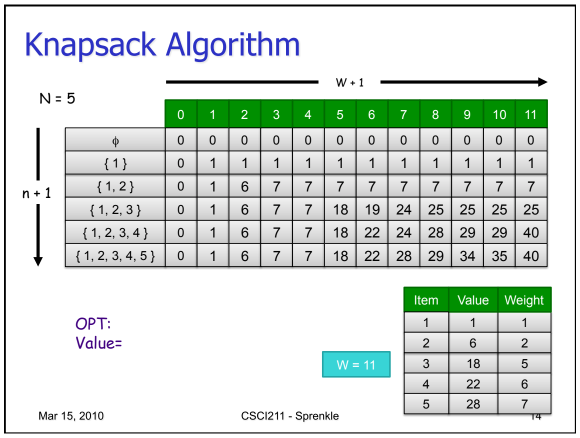
	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0											

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

OPT:
Value=

Mar 15, 2010 CSCI211 - Sprenkle 13



Knapsack Algorithm

N = 5

W + 1 →

n + 1 ↓

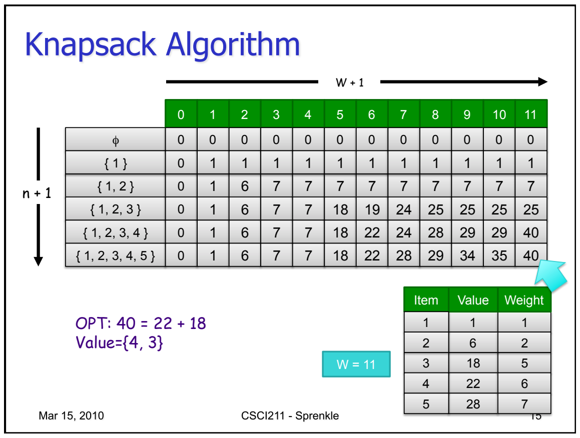
	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

OPT:
Value=

Mar 15, 2010 CSCI211 - Sprenkle 14



Knapsack Algorithm

W + 1 →

n + 1 ↓

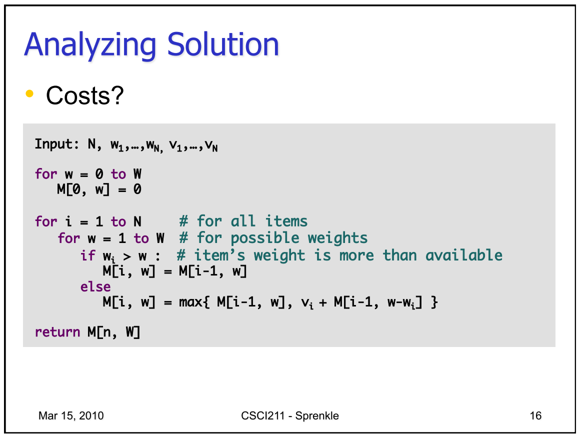
	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

OPT: 40 = 22 + 18
Value={4, 3}

Mar 15, 2010 CSCI211 - Sprenkle 15



Analyzing Solution

- Costs?

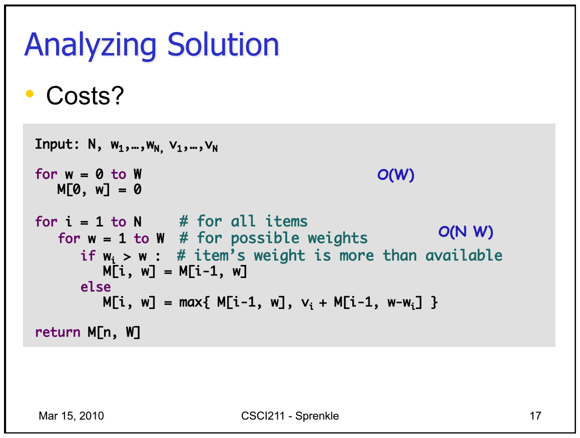
```
Input: N, $w_1,…,w_N$, $v_1,…,v_N$

for w = 0 to W
   M[0, w] = 0

for i = 1 to N     # for all items
   for w = 1 to W  # for possible weights
      if $w_i$ > w :  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], $v_i$ + M[i-1, w-$w_i$] }

return M[n, W]
```

Mar 15, 2010 CSCI211 - Sprenkle 16



Analyzing Solution

- Costs?

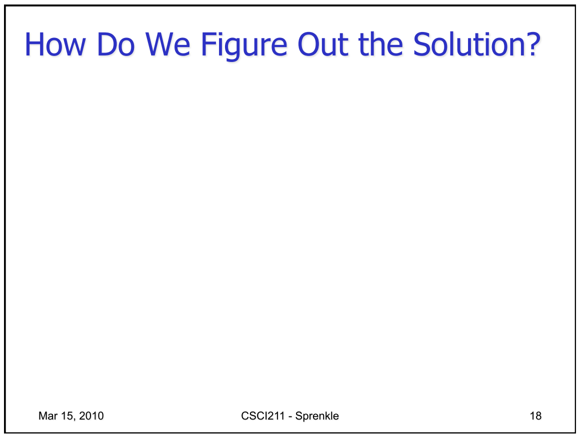
```
Input: N, $w_1,…,w_N$, $v_1,…,v_N$

for w = 0 to W                              O(W)
   M[0, w] = 0

for i = 1 to N     # for all items
   for w = 1 to W  # for possible weights           O(N W)
      if $w_i$ > w :  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], $v_i$ + M[i-1, w-$w_i$] }

return M[n, W]
```

Mar 15, 2010 CSCI211 - Sprenkle 17



How Do We Figure Out the Solution?

Mar 15, 2010 CSCI211 - Sprenkle 18

Knapsack Problem: Running Time

- Running time. Θ(n W)
  - **Not** polynomial in input size!
  - "Pseudo-polynomial"
    - Reasonably efficient when W is reasonably small
  - Decision version of Knapsack is NP-complete [Chapter 8]
- Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Review: Dynamic Programming

- What is the key idea?
- What is our approach to solve a problem using dynamic programming?

Review: Dynamic Programming

- What is the key idea?
  - Memoization: remember the answer for subproblems
    - Improves running time
    - Tradeoff in space
- What is our approach to solve a problem using dynamic programming?
  - Figure out what we're optimizing
  - Figure out how to break the problem into subproblems
  - Figure out how to compute solution from subproblems
  - Define the recurrence relation between the problems

What was the Key to Solving each of these Problems?

- Weighted interval scheduling
- Segmented least squares
- Knapsack

What was the Key to Solving each of these Problems?

- Weighted interval scheduling
  - Binary decision: job was in or wasn't
  - Know conflicts→ reduce problem
- Segmented least squares
  - Knew last point was definitely in one segment
    - Could reduce
  - Multiway decision→ many possibilities for segment starting point
- Knapsack
  - If select an item, reduce available size by item's size
    - Find opt solution for smaller weight, remaining items

This Week

- Wed: Wiki
  - Chapter 6, up to and including 6.4
- Friday: Problem Set 7 due
  - Exam 2 will be handed out