Objectives

- Dynamic Programming
  - Knapsack

Review

- What is the segmented least squares problem?
- What was our solution?

Segmented Least Squares: Algorithm Analysis

How do we find the *solution*?

```
INPUT: n, p1,…,pn , c

Segmented-Least-Squares()
   M[0] = 0
   e[0][0] = 0
   for j = 1 to n
      for i = 1 to j
         e[i][j] = least square error for the
                   segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (e[i][j] + c + M[i-1])

   return M[n]
```

can be improved to $O(n^2)$ by pre-computing various statistics

$O(n^3)$

$O(n^2)$

- Bottleneck: computing e(i, j) for $O(n^2)$ pairs, O(n) per pair using previous formula

Post-Processing: Finding the Solution

```
FindSegments(j):
   if j = 0:
      output nothing
   else:
      Find an i that minimizes ei,j + c + M[i-1]
      Output the segment {pi, …, pj}
      FindSegments(i-1)
```

Cost? $O(n^2)$

Call as: `FindSegments(n)`

Review

- What is the knapsack problem?
- What is the logic behind our solution so far?

Greedy Solution Won't Work

- Good to try greedy solution first:
  - Typically fast, straightforward algorithm
- Greedy idea: order by value/weight
  - Issue: not infinite supply of items
  - Counterexample:
    - Weight of knapsack: 20

Item	A	B	C	D
Value	35	90	60	90
Size	5	15	10	10
Ratio	7	6	6	9

Greedy: D, A = 125
Optimal: D, C = 150

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*?
  - Either select item *i* or not
  - If don't select
    - Pick optimum solution of remaining items
  - Otherwise

What happens?
How does problem change?
Formulate the recurrence

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 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: W, N, w1,…,wN, v1,…,vN

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

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

return M[N, W]
```


Knapsack Input

W = 11

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

Knapsack Algorithm

Represents weight in knapsack

W → # of entries: W + 1

i

# of entries: 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											

Represents item id

W = 11

OPT:
Solution =

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

Knapsack Algorithm

W + 1

i = 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:
Solution =

W = 11

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

Knapsack Algorithm

i = 2

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, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

OPT:
Solution =

W = 11

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

Knapsack Algorithm

i = 3

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, 2, 3, 4, 5 }	0											

OPT:
Solution =

W = 11

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

Knapsack Algorithm

i = 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											

OPT:
Solution =

W = 11

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

Knapsack Algorithm

i = 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

OPT:
Solution =

W = 11

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

What is the optimal solution?

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

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

W = 11

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

Analyzing Solution

How do we figure out the optimal *solution*?

```
Input: W, N, w1,…,wN, v1,…,vN

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

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

return M[n, W]
```

Costs?

Analyzing Solution

```
Input: W, N, w1,…,wN, v1,…,vN

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

return M[n, W]
```


Knapsack Problem: Running Time

- Running time. $\Theta(n\ W)$
  - **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]

Looking Ahead

- Wiki – 6 – 6.3
- PS8 – Due Friday