

**CS 401**

**Dynamic Programming**

Xiaorui Sun

# Dynamic Programming

## Principle:

- Optimal substructure: Remove certain part of the optimal solution (for the entire problem) is an optimal solution of a subproblem
- Typically, only a polynomial number of subproblems

## Technique:

- Parameterization: Describe subproblems by parameters so that the optimal solution can be represented as a recurrence relation
- Memorization: Remember the solution of subproblems

## Examples:

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Multidimensional dynamic programming: knapsack

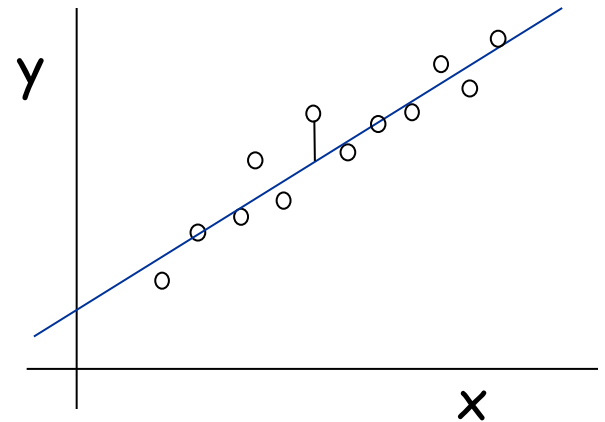
# Segmented Least Squares

# Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- Given  $n$  points in the plane:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ .
- Find a line  $y = ax + b$  that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^n (y_i - ax_i - b)^2$$



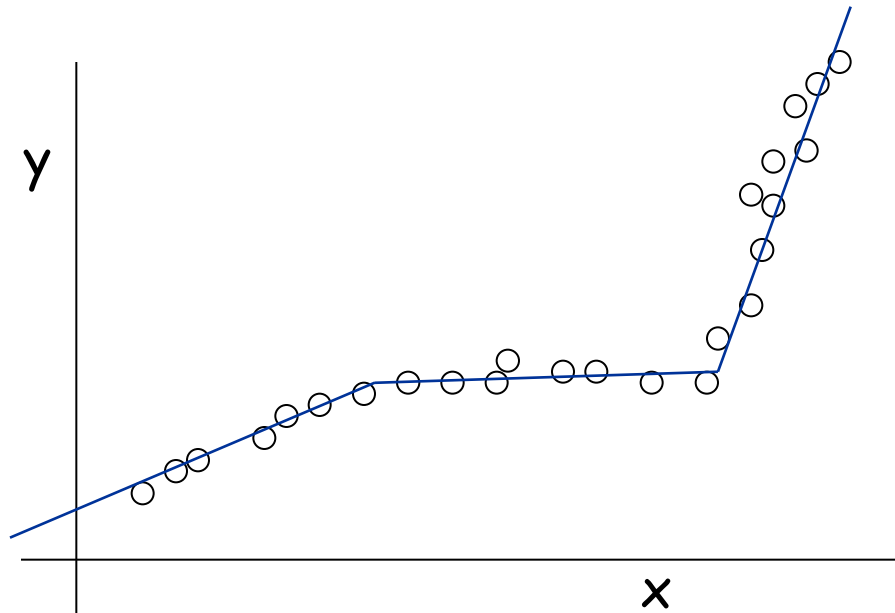
Solution. Calculus  $\Rightarrow$  min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i) (\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

# Segmented Least Squares

Segmented least squares.

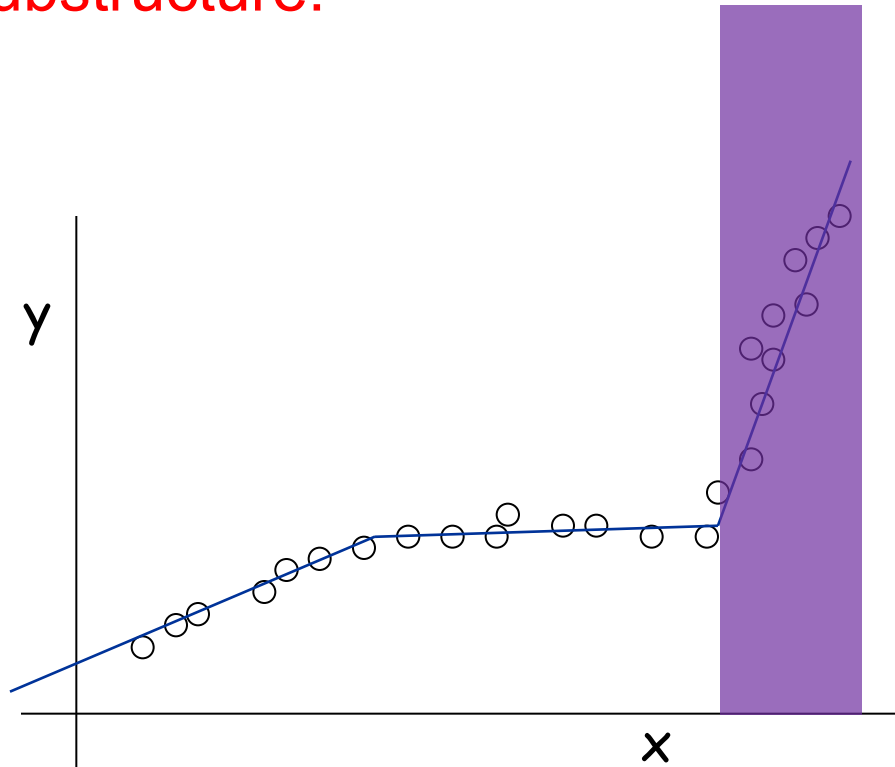
- Points lie roughly on a sequence of several line segments.
- Given  $n$  points in the plane  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  with
- $x_1 < x_2 < \dots < x_n$ , find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors  $E$  in each segment
  - the number of lines  $L$
- Tradeoff function:  $E + cL$ , for some constant  $c > 0$ .



# Dynamic programming

Suppose we know the last segment

- If all the points in last segment are removed, then the remaining segments must be the optimal solution for the remaining points
- **Optimal substructure!**



# Dynamic Programming: Multiway Choice

Notation.

$OPT(j)$  = minimum cost for points  $p_1, \dots, p_{i+1}, \dots, p_j$ .

$e(i, j)$  = minimum sum of squares for points  $p_i, p_{i+1}, \dots, p_j$ .

To compute  $OPT(j)$ :

Last segment uses points  $p_i, p_{i+1}, \dots, p_j$  for some  $i$ .

Cost =  $e(i, j) + c + OPT(i-1)$ .

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e(i, j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

# Segmented Least Squares: Algorithm

**INPUT:**  $n, p_1, \dots, p_N, c$

```
Segmented-Least-Squares() {  
  M[0] = 0  
  for j = 1 to n  
    for i = 1 to j  
      compute the least square error  $e_{ij}$  for  
      the segment  $p_i, \dots, p_j$   
  
  for j = 1 to n  
    M[j] =  $\min_{1 \leq i \leq j} (e_{ij} + c + M[i-1])$   
  
  return M[n]  
}
```

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

Running time.  $O(n^3)$ .

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



# Knapsack Problem

# Knapsack Problem



Given  $n$  objects and a "knapsack."

Item  $i$  weighs  $w_i > 0$  kilograms and has value  $v_i > 0$ .

Knapsack has capacity of  $W$  kilograms.

**Goal:** fill knapsack so as to maximize total value.

**Ex:** OPT is { 3, 4 } with value 40.

$$W = 11$$

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

**Greedy:** repeatedly add item with maximum ratio  $v_i/w_i$ .

**Ex:** { 5, 2, 1 } achieves only value = 35  $\Rightarrow$  greedy not optimal.

# First Attempt

Def  $OPT(i) = \text{max profit subset of items } 1, \dots, i.$

Case 1:  $OPT$  does not select item  $i$ .

- $OPT$  selects best of  $\{ 1, 2, \dots, i - 1 \}$

Case 2:  $OPT$  selects item  $i$ .

- accepting item  $i$  does not immediately imply that we will have to reject other items
- without knowing what other items were selected before  $i$ , we don't even know if we have enough room for  $i$

# Stronger DP (New Variable)

Let  $OPT(i, w)$  = Max value of subsets of items  $1, \dots, i$  of weight  $\leq w$

**Case 1:**  $OPT(i, w)$  selects item  $i$

- In this case,  $OPT(i, w) = v_i + OPT(i - 1, w - w_i)$

**Case 2:**  $OPT(i, w)$  does not select item  $i$

- In this case,  $OPT(i, w) = OPT(i - 1, w)$ .

Take best of the two



Therefore,

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

# DP for Knapsack

```
Compute-OPT(i,w)
  if M[i,w] == empty
    if (i==0)
      M[i,w]=0
    else if (wi > w)
      M[i,w]=Comp-OPT(i-1,w)
    else
      M[i,w]= max {Comp-OPT(i-1,w), vi + Comp-OPT(i-1,w-wi) }
  return M[i, w]
```

recursive

```
for w = 0 to W
  M[0, w] = 0
for i = 1 to n
  for w = 1 to W
    if (wi > w)
      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]
```

Non-recursive

# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	$\phi$	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											

$W = 11$

```

if ( $w_i > w$ )
     $M[i, w] = M[i-1, w]$ 
else
     $M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i ]\}$ 
    
```

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

# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	$\phi$	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											

$W = 11$

```

if (wi > w)
    M[i, w] = M[i-1, w]
else
    M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}
    
```

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

# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	$\phi$	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								
	{1, 2, 3}	0	1										
	{1, 2, 3, 4}	0	1										
	{1, 2, 3, 4, 5}	0	1										

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

$W = 11$

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

```

if (wi > w)
    M[i, w] = M[i-1, w]
else
    M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}
    
```



# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	$\phi$	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					
	{1,2,3,4}	0	1										
	{1,2,3,4,5}	0	1										

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

$W = 11$

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

```

if (wi > w)
    M[i, w] = M[i-1, w]
else
    M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}
    
```

# DP for Knapsack

←————— W + 1 —————→

		0	1	2	3	4	5	6	7	8	9	10	11
<div style="display: flex; align-items: center;"> <div style="border-left: 1px solid black; border-right: 1px solid black; height: 100px; margin-right: 5px;"></div> <div style="display: flex; flex-direction: column; align-items: center; justify-content: space-between; width: 20px;"> <span>n + 1</span> <span>↓</span> </div> </div>	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		
	{ 1, 2, 3, 4, 5 }	0	1										

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

W = 11

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

```

if (wi > w)
    M[i, w] = M[i-1, w]
else
    M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
    
```

# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	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	34	40

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

$W = 11$

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

```

if (wi > w)
    M[i, w] = M[i-1, w]
else
    M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}
    
```

# DP Ideas so far

- You may have to strengthen DP, equivalently the induction, i.e., you may have to carry more information to find the Optimum.
- This means that sometimes we may have to use two dimensional or three dimensional induction
- Is this dynamic algorithm a polynomial time algorithm for knapsack?
  - No
  - For an input of  $N$  bits,  $W$  can be as large as  $2^N$
  - Knapsack problem is actually NP-complete