Objectives

- Dynamic Programming
  - Review: Weighted Interval Scheduling
  - Wrap up: Least Segmented Squares
  - Knapsack

Review

- What is the new algorithm design technique we're learning?
  - What questions do we ask to solve the problems?
  - What is the process? What are the components of the algorithm?
- What problems have we considered so far?

Review Solving Dynamic Programming Problems

1. Determine optimal substructure of problem
   - Define the recurrence relation
2. Define algorithm to find the ***value*** of optimal solution
3. Optionally, change algorithm to an ***iterative*** rather than recursive solution
4. Define algorithm to find ***optimal solution***
5. Analyze running time of algorithms

Map to weighted-interval scheduling

Review

- What is the segmented least squares problem?

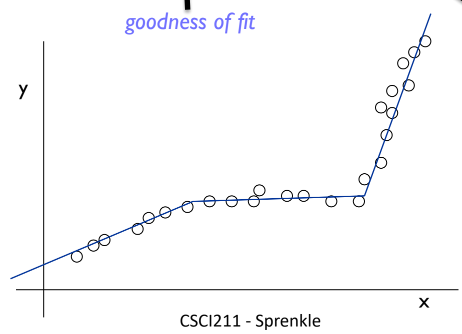
Segmented Least Squares

- Points lie roughly on a **sequence** of line segments
- Given *n* points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of line segments that **minimizes f(x)**

What's a reasonable choice for *f(x)* to balance *accuracy* and *parsimony*?

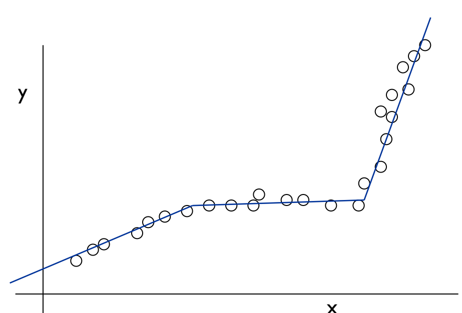
goodness of fit

number of lines



Segmented Least Squares

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

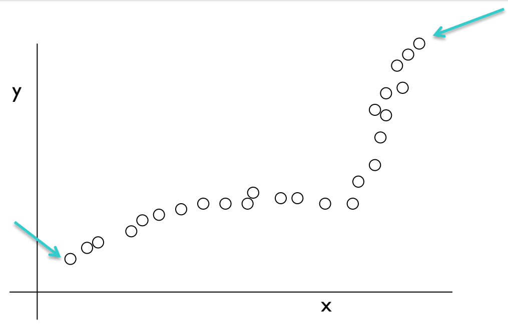


How should we define an optimal solution?

Toward a Solution

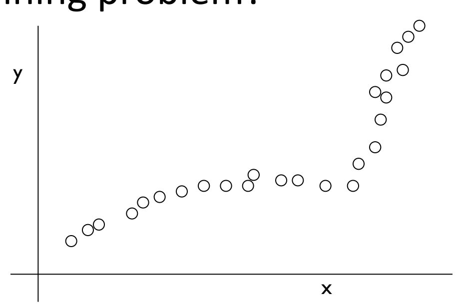
- Consider just the first or last point

What do we know about those points? their segments? cost of a segment?



Toward a Solution

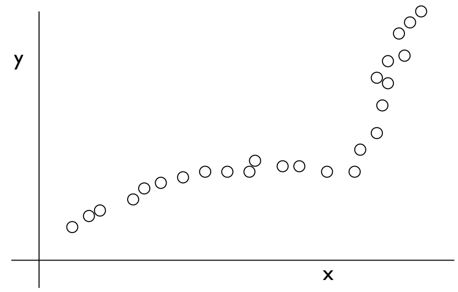
- p_n can only belong to one segment
  - Segment: p_i, …, p_n
  - Cost: c (cost for segment) + error of segment
- What is the remaining problem?



Toward a Solution

- p_n can only belong to one segment
  - Segment: $p_i, \ldots, p_n$
  - Cost: c (cost for segment) + error of segment
- What is the remaining problem?
  - Solve for $p_1, \ldots, p_{i-1}$

Next: Formulate as a recurrence



Dynamic Programming: Multiway Choice

- Notation.
  - **OPT(j)** = minimum cost for points $p_1, p_2, \ldots, p_j$.
  - **e(i, j)** = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- How do we compute OPT(j)?
  - Last problem: binary decision (include job or not)
  - This time: **multiway** decision
    - Which option do we choose?

Dynamic Programming: Multiway Choice

- Notation.
  - **OPT(j)** = minimum cost for points $p_1, p_2, \ldots, p_j$.
  - **e(i, j)** = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute OPT(j):
  - Last segment contains points $p_i, p_{i+1}, \ldots, p_j$ for some i
  - Cost = e(i, j) + c + OPT(i-1).

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

Mar 22, 2019 CSCI211 - Sprenkle 11

Segmented Least Squares: Algorithm Analysis

Costs?

```
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]
```

Mar 22, 2019 CSCI211 - Sprenkle 12

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?

Call as: `FindSegments(n)`

Post-Processing: Finding the Solution

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

Cost? $O(n^2)$

Call as: `FindSegments(n)`

KNAPSACK

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

Greedy Solution Won't Work

- Should 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

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?
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

```
Input: W, N, w1,…,wN, v1,…,vN

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

for i = 1 to N
   for w = 0 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]
```


Knapsack Problem: Bottom-Up

- Fill up an n-by-W array

```
Input: W, N, w1,…,wN, v1,…,vN

for w = 0 to W    # base case: no items, so value is 0
   M[0, w] = 0

for i = 1 to N     # for all items
   for w = 0 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]
```

Looking Ahead

- Wiki due Monday
  - Chap 6: 6.1-6.4
- PS8 due Friday