Objectives

- Dynamic Programming
  - Segmented Least Squares
  - Subset Sums/Knapsacks

Review: Weighted Interval Scheduling

- What was the key insight to solving the weighted interval scheduling problem?

Binary decision:
- Optimal solution for jobs 1 through j includes j or doesn't

- How do we pick the solution?

Choose the larger value of
- [choose j and the best solution of compatible jobs] OR [best solution if don't pick j]

Review: Iterative Solution to find the Optimal Value

- Build up solution from subproblems instead of breaking down

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt:
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
```

- Typically, approach we'll take

Review: Finding the Solution

- Dynamic programming algorithms compute optimal value.
- What if we want the ***solution*** itself (**not** simply the value)?
- Do some post-processing

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j):
   if j = 0:
      output nothing
   elif vj + M[p(j)] > M[j-1]:
      print j
      Find-Solution(p(j))
   else:
      Find-Solution(j-1)
```


Review: Dynamic Programming Process

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

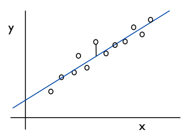
Map to weighted interval scheduling problem

SEGMENTED LEAST SQUARES

Least Squares

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

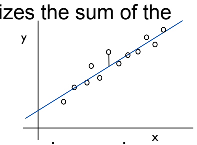
Sum of squared error $SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$



Least Squares

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

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

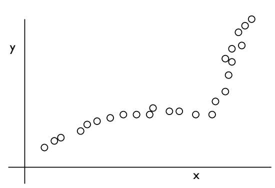


- Closed form solution. Calculus $\Rightarrow$ min error is achieved when

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

Least Squares

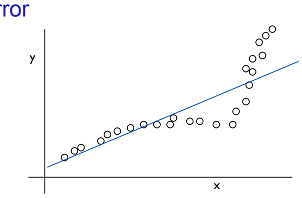
- What happens to the error if we try to fit one line to these points?



- What pattern does it seem like these points have?

Least Squares

- What happens to the error if we try to fit one line to these points?
  - Large error

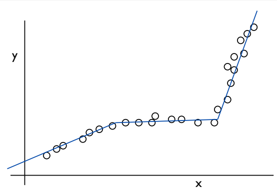


- Pattern: More like 3 lines

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)**

If I want the ***best*** fit, how many lines should I use?



Segmented Least Squares

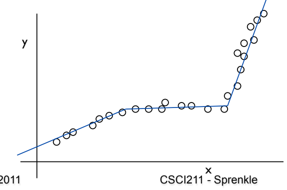
- Points lie roughly on a **sequence** of line segments
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of line segments that **minimizes f(x)**

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

goodness of fit *number of lines*

Segmented Least Squares

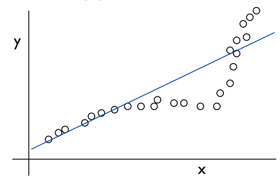
- Points lie roughly on a **sequence** of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of line segments that minimizes:
  - E: sum of the sums of the squared errors in each segment
  - L: the number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



How should we define an optimal solution?

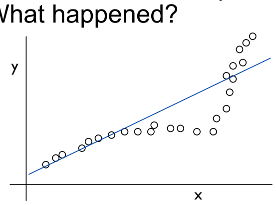
Segmented Least Squares

- What made it seem like the points were in 3 lines? What happened?



Segmented Least Squares

- What made it seem like the points were in 3 lines? What happened?



- Error increased
- Looking for *change* in linear approximation
  - Where to partition points into line segments

Recall: Properties of Problems for DP

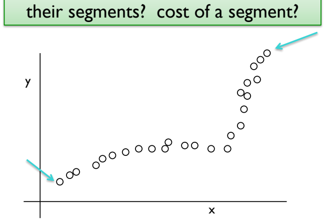
- Polynomial number of subproblems
- Solution to original problem can be easily computed from solutions to subproblems
- Natural ordering of subproblems, easy to compute recurrence

We need to:
- Figure out how to break the problem into subproblems
- Figure out how to compute solution from subproblems
- Define the recurrence relation between the problems

Toward a Solution

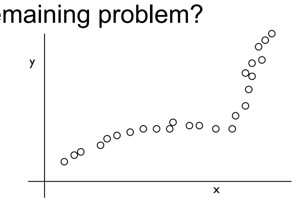
- Consider just the first or last point

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



Toward a Solution

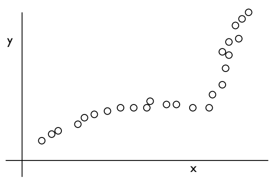
- p_n can only belong to one segment
  - Segment: $p_i, \ldots, p_n$
  - Cost: c (cost for segment) + error of segment
- What is the remaining problem?



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 $p_1, p_{i+1}, \ldots, p_j$.
  - **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 $p_1, p_{i+1}, \ldots, p_j$.
  - **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).

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

Segmented Least Squares: Algorithm

```
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]
```

Costs?

Segmented Least Squares: Algorithm Analysis

```
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

How Do We Find the *Solution*?

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)

SUBSET SUMS and 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 $v_i > 0$
  - Alternative: jobs require w_i time
- Knapsack has capacity of *W* kilograms
  - Alternative: *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

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

Knapsack Problem: Bottom-Up

- Fill up an n-by-W array

```
Input: N, 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 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 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											

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

OPT:
Value=

W = 11

Assignments

- Continue reading Chapter 6
- Wiki for Wednesday
  - 5.5, 6 front matter, 6.1-6.4
- PS7 due Friday
- Next Wednesday, 4 p.m.: talk by Jan Cuny about broadening participation in computing
  - Last extra credit opportunity of the term