Objectives

- Dynamic Programming
  - Wrapping up: weighted interval schedule
  - Segmented Least Squares

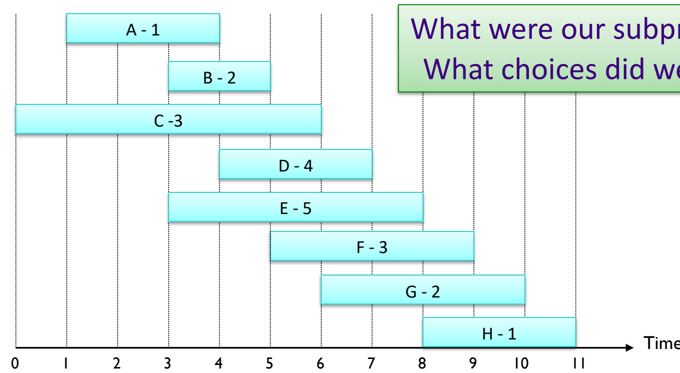
Summary: Properties of Problems for Dynamic Programming

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

Get out handouts from last time…

Review: Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j, and has weight or value v_j
- Two jobs are compatible if they don't overlap
- **Goal**: find maximum weight subset of mutually compatible jobs



What were our subproblems?
What choices did we make?

Weighted Interval Scheduling: Memoization Analysis

Costs?

```
Input: n jobs (associated start time s_j, finish time f_j, and
value v_j)

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j):
   if M[j] is empty:
      M[j] = max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]

M-Compute-Opt(n)
```

Weighted Interval Scheduling: Memoization Analysis

```
Input: n jobs (associated start time sj, finish time fj, and
value vj)

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn   O(n log n)
Compute p(1), p(2), …, p(n)  O(n log n);

for j = 1 to n
   M[j] = empty    O(n)
M[0] = 0

M-Compute-Opt(j):
   if M[j] is empty:
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]

M-Compute-Opt(n)  O(n)
```


Weighted Interval Scheduling: Running Time

- Claim. Memoized version of algorithm takes O(n log n) time
  - Sort by finish time: O(n log n)
  - Computing p(·): O(n log n)
  - `M-Compute-Opt(j)`: each invocation takes O(1) time and either
    - (i) returns an existing value `M[j]`
    - (ii) fills in one new entry `M[j]` and makes two recursive calls
  - Progress measure Φ = # nonempty entries of `M[]`
    - (i) initially $\Phi = 0$, throughout $\Phi \le n$
    - (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls
  - Running time of `M-Compute-Opt(n)` is O(n). ▪
- Remark.
  - O(n) if jobs are *pre-sorted* by start and finish times – see textbook

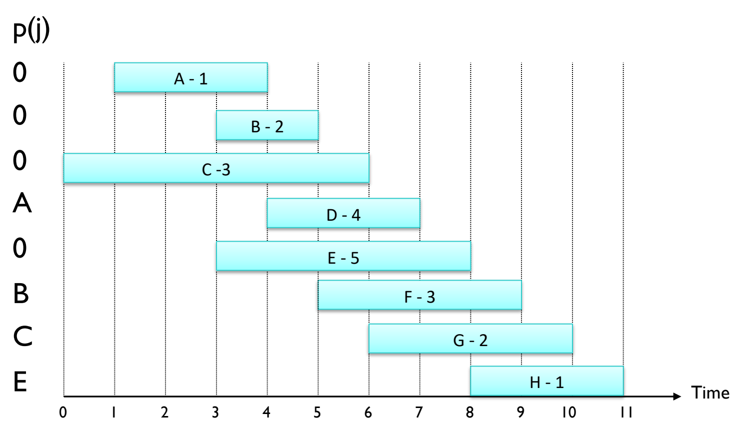
Weighted Interval Scheduling: Finding a Solution

- Dynamic programming algorithms compute ***optimal value***
- What if we want the ***solution*** itself?
  - **Not** simply the optimal value
- Do some post-processing
  - Looking at M, how do we know which set of intervals were chosen?

M

0	A	B	C	D	E	F	G	H
0	**1**	**2**	**3**	**5**	**5**	**5**	**5**	**6**

Towards Finding a Solution



M

0	A	B	C	D	E	F	G	H
0	**1**	**2**	**3**	**5**	**5**	**5**	**5**	**6**

Weighted Interval Scheduling: Finding a Solution

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

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

def Find-Solution(j):
    if j = 0:
        output nothing
    elif did I pick the job?:
        print j
        Find-Solution(p(j))
    else:
        Find-Solution(j-1)
```


Weighted Interval Scheduling: Finding a Solution

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

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

def 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)
```

Runtime?

O(n)

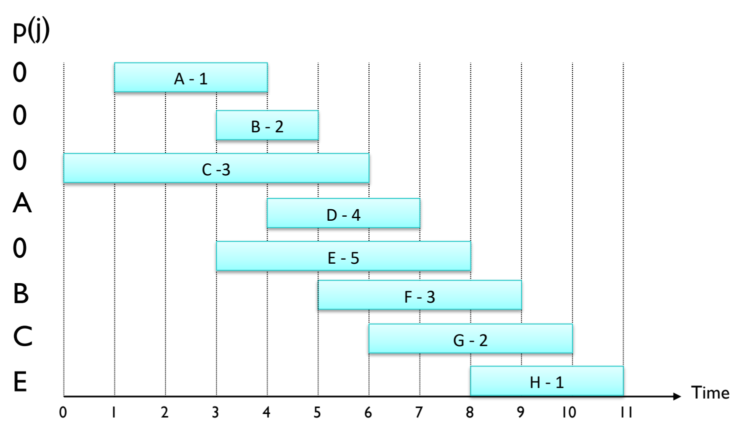
Turning it Around…

- We solved as a recursive/memoized algorithm

Can we write this algorithm as an **iterative** solution?

```
Input: n jobs (associated start time sj, finish time fj, and
value vj)

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j):
   if M[j] is empty:
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
M-Compute-Opt(n)
```

Towards Iterative Solution…



M	**0**	A	B	C	D	E	F	G	H

Iterative Solution

- Build up solution from subproblems instead of breaking down

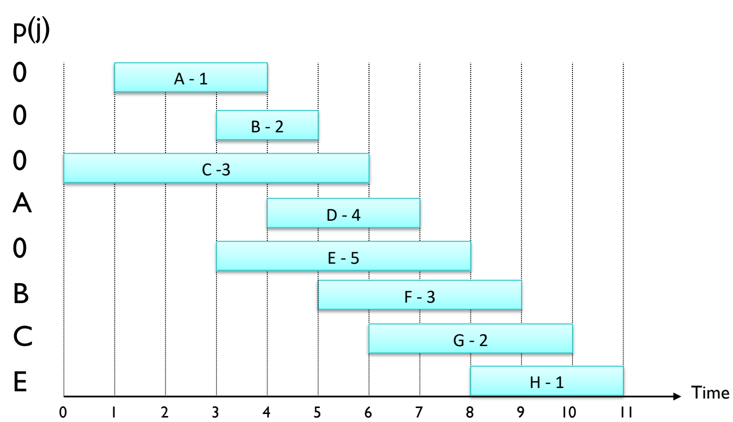
```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

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

Compute p(1), p(2), …, p(n)

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

Runtime?

O(n)

- Typically, we'll take iterative approach

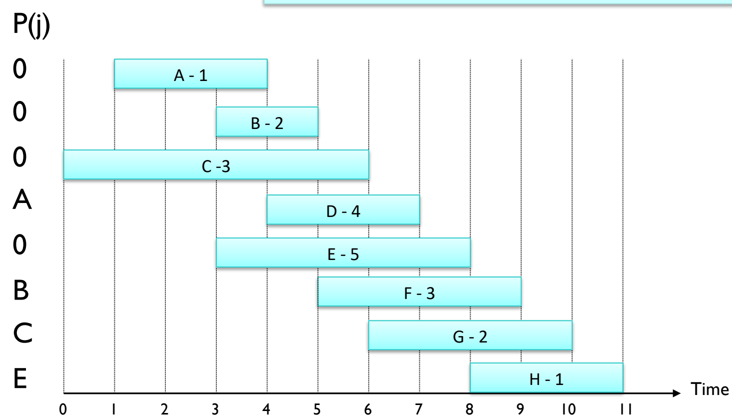
Example: Iteratively



M	0	A	B	C	D	E	F	G	H
	0								

Example: Iteratively

M[j] = max(v_j + M[p(j)], M[j-1])

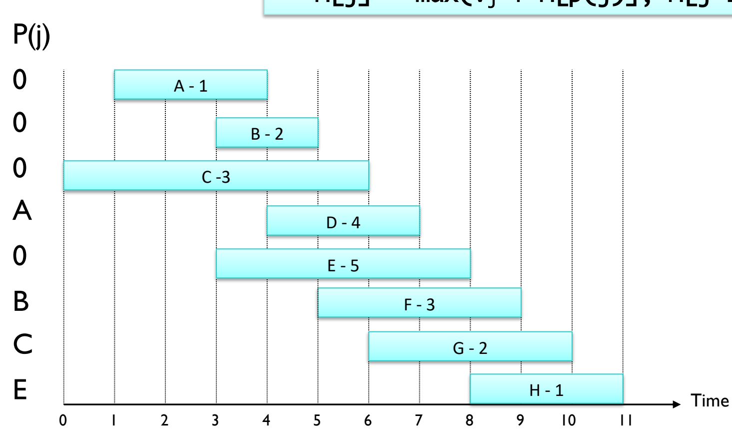


M

0	A	B	C	D	E	F	G	H
0								

Example: Iteratively

M[j] = max(v_j + M[p(j)], M[j-1])

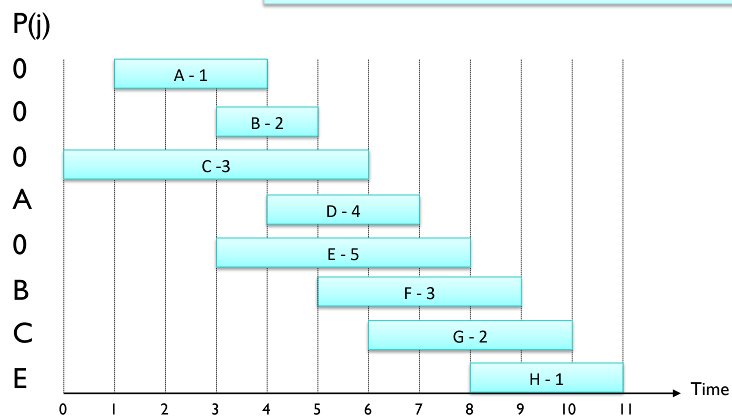


M

0	A	B	C	D	E	F	G	H
0	**1**							

Example: Iteratively

`M[j] = max(v_j + M[p(j)], M[j-1])`

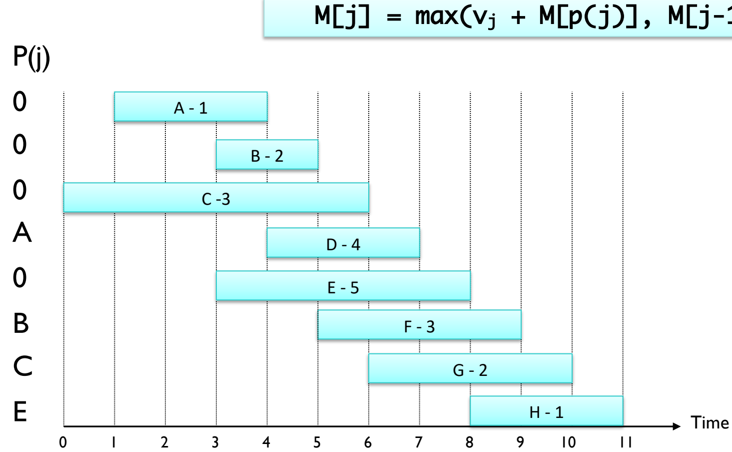


M

0	A	B	C	D	E	F	G	H
0	1							

Example: Iteratively

`M[j] = max(v_j + M[p(j)], M[j-1])`

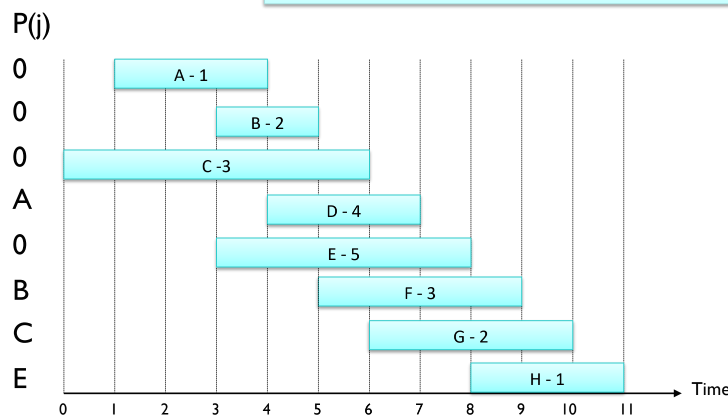


M

0	A	B	C	D	E	F	G	H
0	1	2						

Example: Iteratively

`M[j] = max(v_j + M[p(j)], M[j-1])`

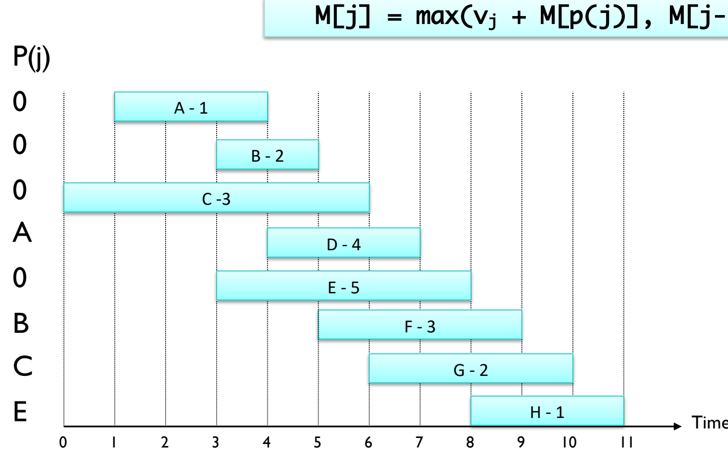


M

0	A	B	C	D	E	F	G	H
0	1	2	3					

Example: Iteratively

`M[j] = max(v_j + M[p(j)], M[j-1])`

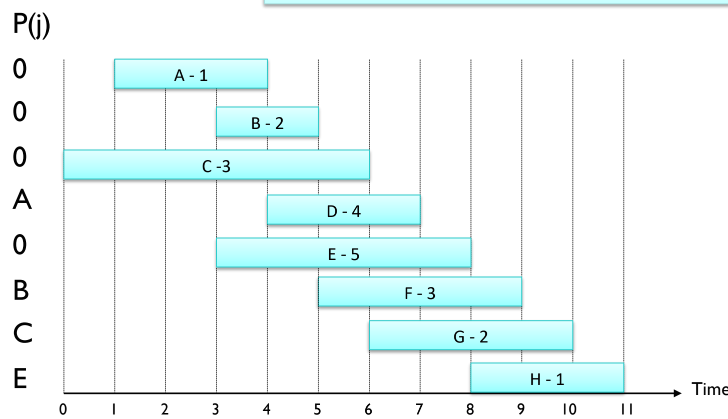


M

0	A	B	C	D	E	F	G	H
0	1	2	3	5				

Example: Iteratively

`M[j] = max(vj + M[p(j)], M[j-1])`



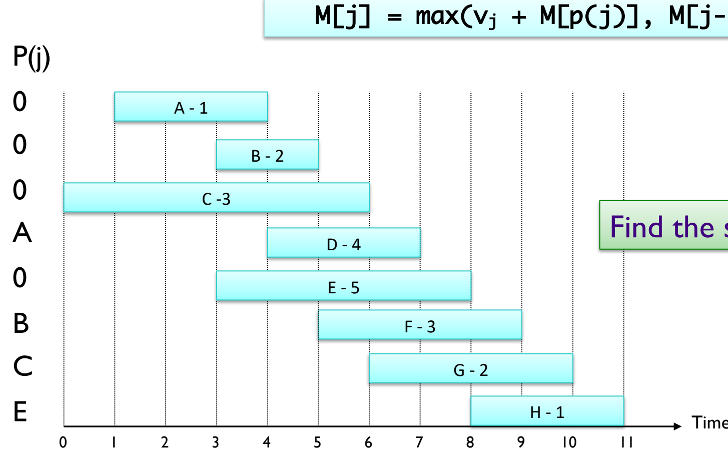
M

0	A	B	C	D	E	F	G	H
0	**1**	**2**	**3**	**5**				

And so on….

Example: Iteratively

`M[j] = max(vj + M[p(j)], M[j-1])`



Find the solution?

M

0	A	B	C	D	E	F	G	H
0	**1**	**2**	**3**	**5**	**5**	**5**	**5**	**6**

Putting It All Together

```
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)

M[0] = 0
for j = 1 to n
    M[j] = max(vj + M[p(j)], M[j-1])

Find-Solution(n)
```

```
def 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)
```

Total Runtime: O(n logn)

Review: Solving Dynamic Programming Problems

1. Determine optimal substructure of problem
   - Ask, what is the problem we're solving?
   - 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

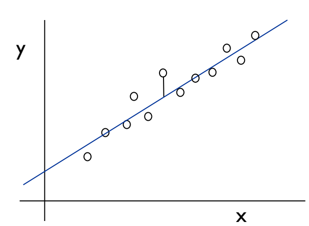
SEGMENTED LEAST SQUARES

Least Squares

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

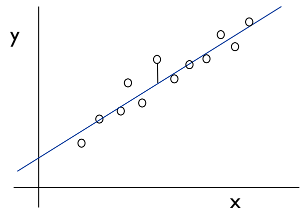
Sum of squared error

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



Least Squares

- Foundational problem in statistics 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

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

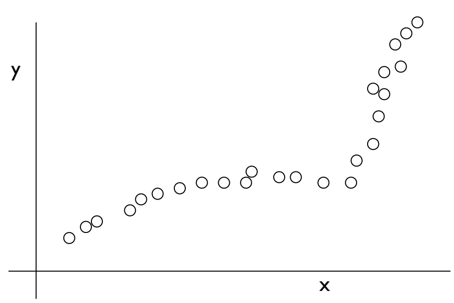
- Closed form solution. Calculus $\Rightarrow$ min error is achieved when

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

Mar 20, 2019 CSCI211 - Sprenkle 27

Least Squares

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

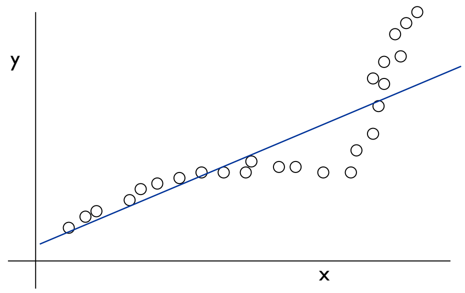


- What pattern does it seem like these points have?

Mar 20, 2019 CSCI211 - Sprenkle 28

Least Squares

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

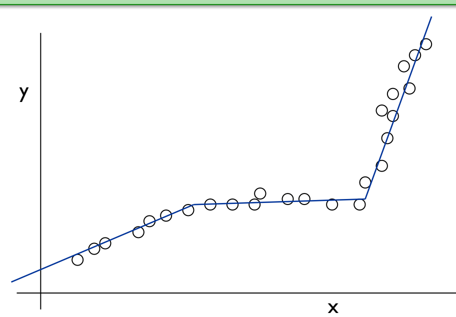


- Pattern: More like 3 lines

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?

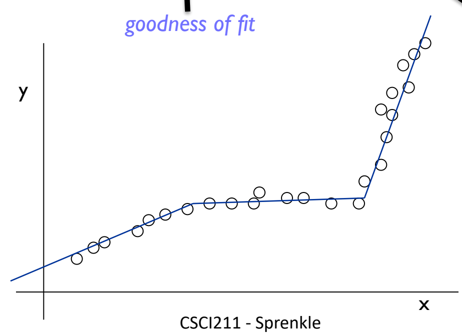


Segmented Least Squares

- 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)**

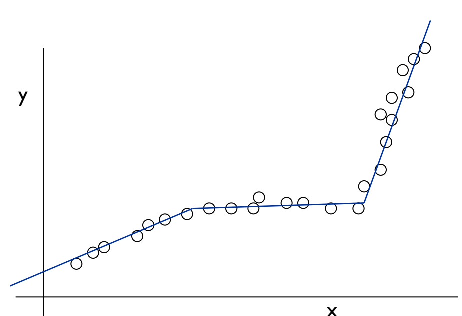
What's a reasonable choice for *f(x)* to balance *accuracy* and *parsimony*?

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?

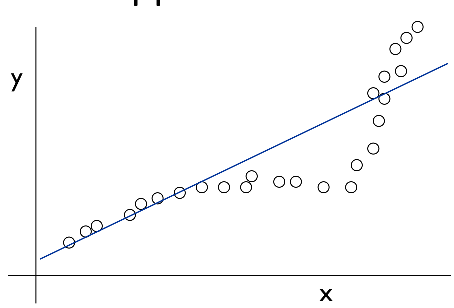
Recall:
Properties of Problems for DP

- 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

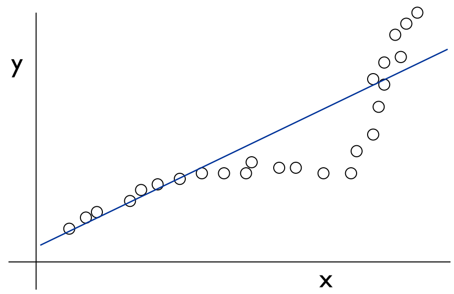
Segmented Least Squares

- 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?

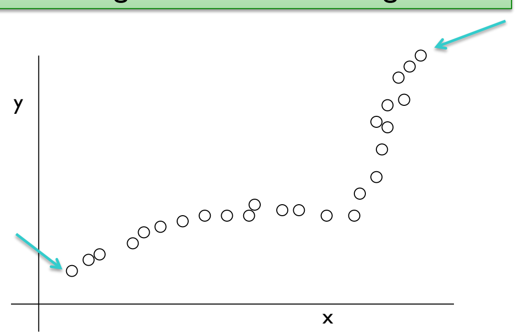


- Error increased
- Looking for *change* in linear approximation
  - Where to partition points into line segments

Toward a Solution

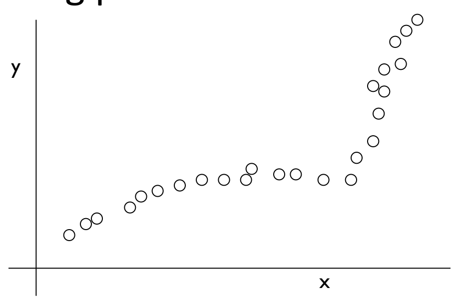
- Consider just the first or last point

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



Toward a Solution

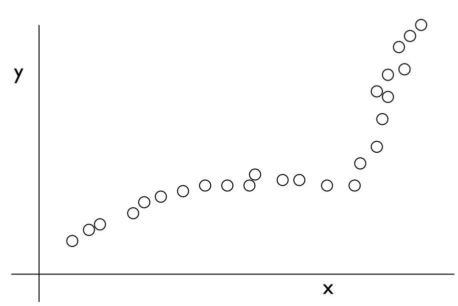
- p_n can only belong to one segment
  - Segment: p_i, …, p_n
  - Cost: c (cost for segment) + error of segment
- What is the remaining problem?



Toward a Solution

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

Next: Formulate as a recurrence



Dynamic Programming: Multiway Choice

- Notation.
  - **OPT(j)** = minimum cost for points p_1, p_{i+1} , … , p_j.
  - **e(i, j)** = minimum sum of squares for points p_i, p_{i+1} , …, 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?

Looking Ahead

- Exam 2 due Friday