Algorithm Design and Analysis

CSE 565

LECTURE 13

Dynamic Programming

- Segmented Least Squares
- Knapsack Problem

Sofya Raskhodnikova

Dynamic Programming

"Those who cannot remember the past are doomed to repeat it."

George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense*

Design Techniques So Far

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- Recursion / divide &conquer. Break up a problem into subproblems, solve subproblems, and combine solutions.
- Dynamic programming. Break problem into *overlapping* subproblems, and build up solutions to larger and larger subproblems.

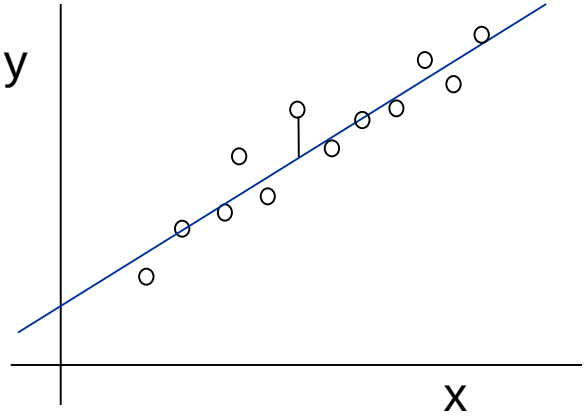
Segmented Least Squares

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:

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



- **Solution.**

  Calculus $\Rightarrow$ min error is achieved when

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

$\Leftarrow$ $O(n)$ time

Segmented Least Squares

Segmented least squares.

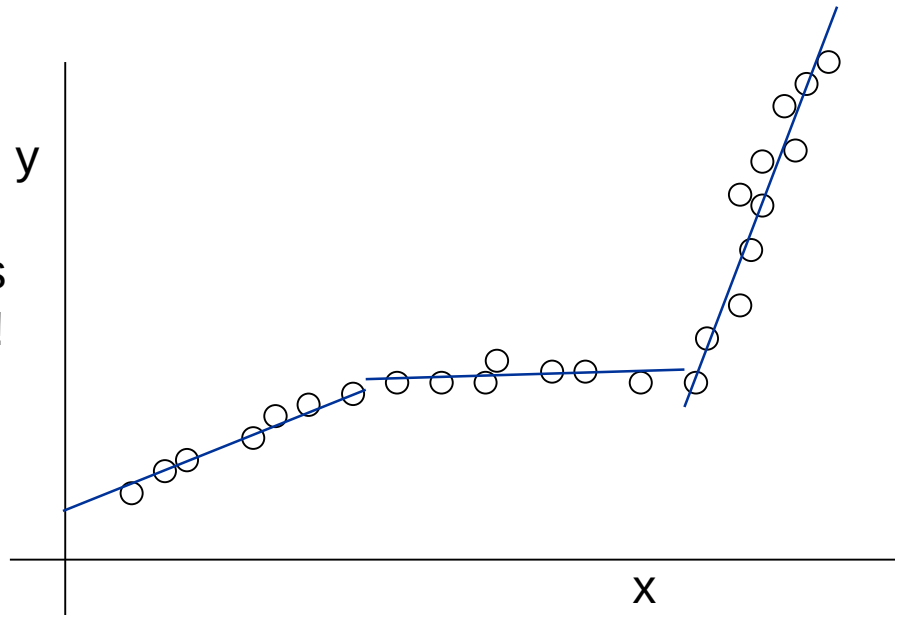
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes f(x).

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

goodness of fit

number of lines

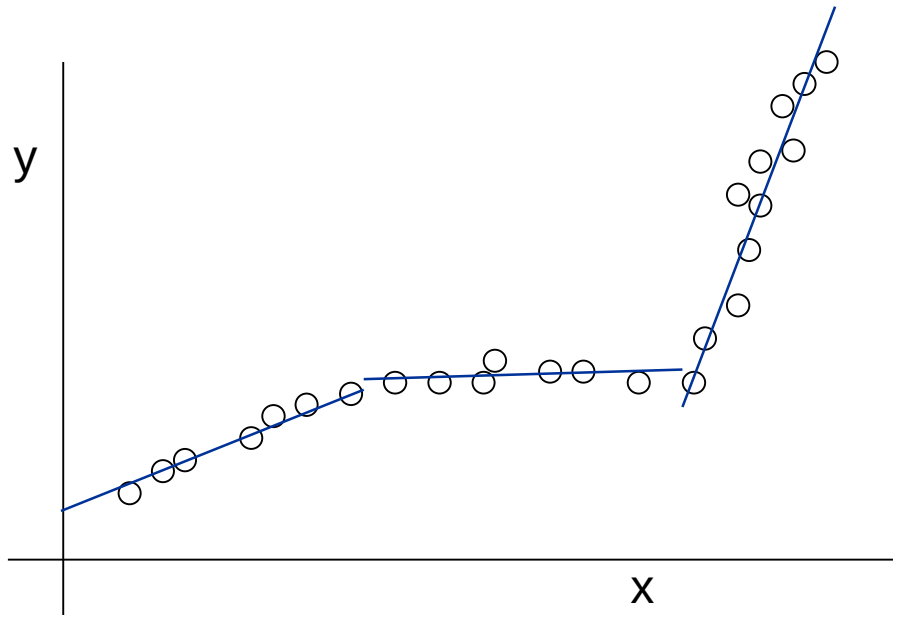
Note: discontinuous functions permitted!



Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, 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: $E + c\,L$, for some constant $c > 0$.



Dynamic Programming: Multiway Choice

Notation.

e(i,j) takes $O(n)$ time to compute

- e(i, j) = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.

To compute OPT(j):

- Last segment uses 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

```
Segmented-Least-Squares(n, p_1, …, p_n, c) {
   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, …, p_j

   for j = 1 to n
      M[j] = min_{1 ≤ i ≤ j} (e_ij + c + M[i-1])

   return M[n]
}
```

Running time. $O(n^3)$. ← Only $O(n^2)$ when $e_{i,j}$'s are precomputed

- 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: { 3, 4 } has value 40.

- Many "packing" problems fit this model
  - Assigning production jobs to a factory
  - Deciding which jobs to do on a single processor with bounded time
  - Deciding which problems to do on an exam

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

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: { 3, 4 } has value 40.
- **Greedy algorithm?**

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

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: { 3, 4 } has value 40.

- **Greedy**: repeatedly add item with maximum ratio v_i / w_i
- Example: { 5, 2, 1 } achieves only value = 35
- Greedy is not optimal.

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Dynamic programming: attempt 1

- Definition: OPT(i) =

  maximum profit on 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.
    - without knowing what other items were selected before i, we don't even know if we have enough room for i

- Conclusion. Need more subproblems!

Adding a new variable

- Definition: OPT(i, w) = max profit on 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 } with weight limit w
  - Case 2: OPT selects item i.
    - new weight limit = $w - w_i$
    - OPT selects best of { 1, 2, …, i–1 } with new weight limit

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

Bottom-up algorithm

- Fill up an n-by-W array.

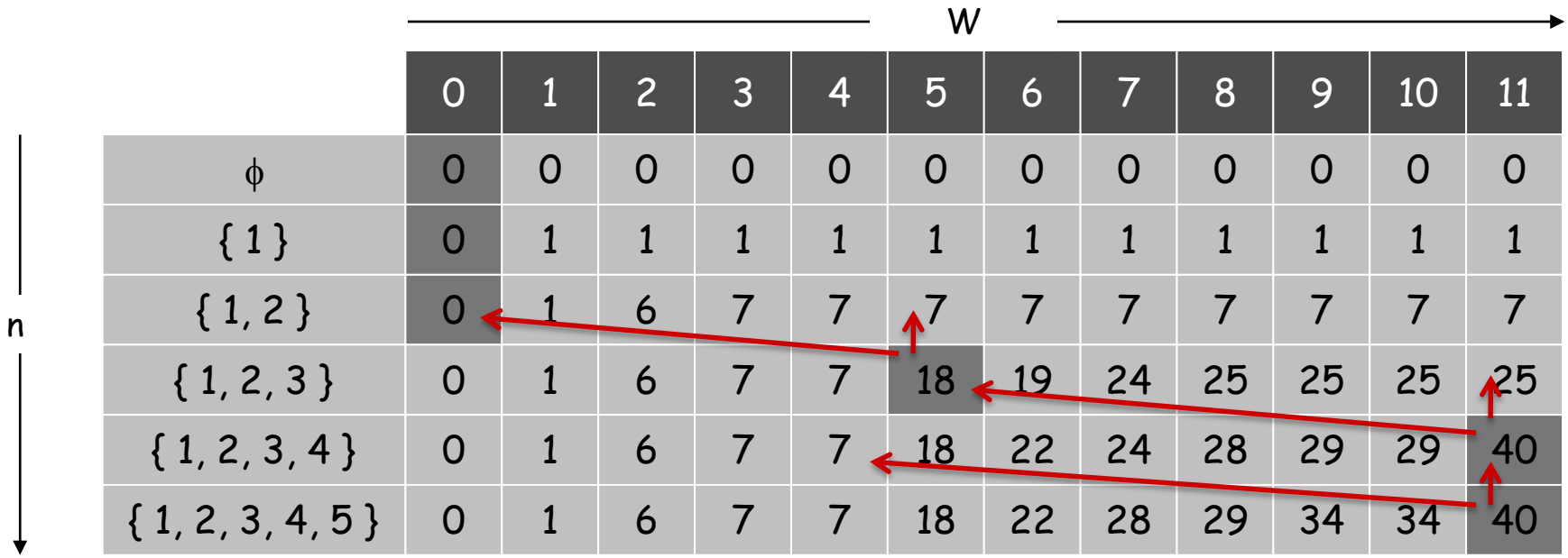
```
Input: n, W, w_1,…,w_N, v_1,…,v_N

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

for i = 1 to n
   for w = 1 to W
      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 ]}

return M[n, W]
```

Knapsack table



	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	34	40

W = 11

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

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

Proof of correctness

- Dynamic programming (and divide and conquer) lends itself directly to induction.
  - Base cases: OPT(i,0)=0, OPT(0,w)=0 (no items!).
  - Inductive step: explaining the correctness of recursive formula
    - If the following values are correctly computed:
      - OPT(0,w-1),OPT(1,w-1),…,OPT(n,w-1)
      - OPT(0,w),…,OPT(i-1,w)
    - Then the recursive formula computes OPT(i,w) correctly
      - Case 1: …, Case 2: … (from previous slides).

Time and space complexity

- 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.

What is the input size?

- In words?
- In bits?

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Time and space complexity

- **Time and space:** $\Theta(n W)$.
  - Not polynomial in input size!
  - Space can be made O(W) (exercise!)
  - "Pseudo-polynomial."
  - Decision version of Knapsack is NP-complete.

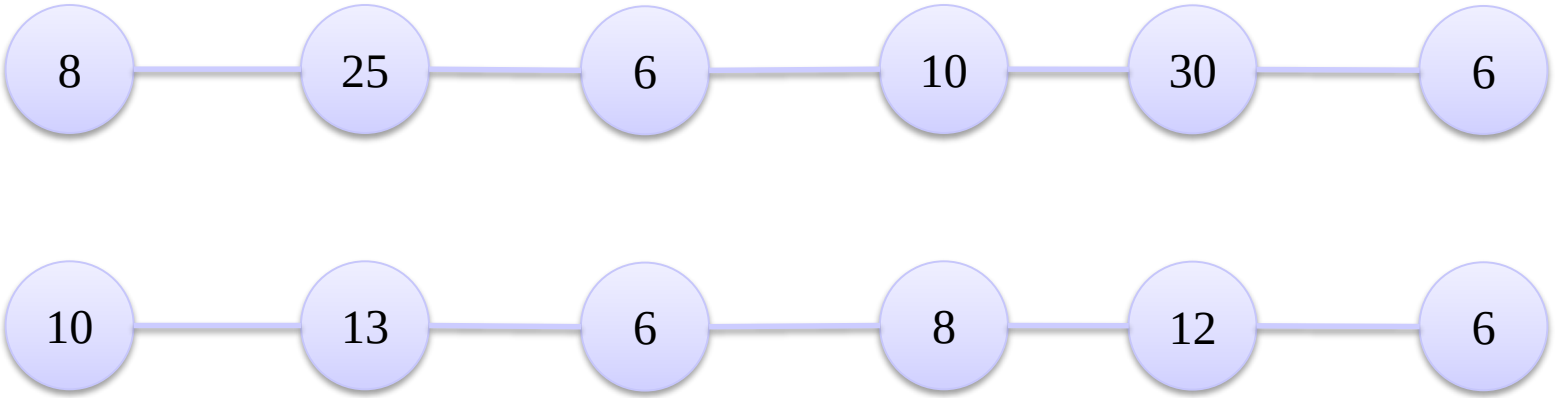
[KT, chapter 8]

- **Knapsack approximation algorithm.** There is a poly-time algorithm that produces a solution with value within 0.01% of optimum. [KT, section 11.8]

Exercise: Weighted Independent Set on a line

Max-weighted IS on a line

- Given a chain of length n and a weight for each vertex, find independent set of maximum weight
- Examples



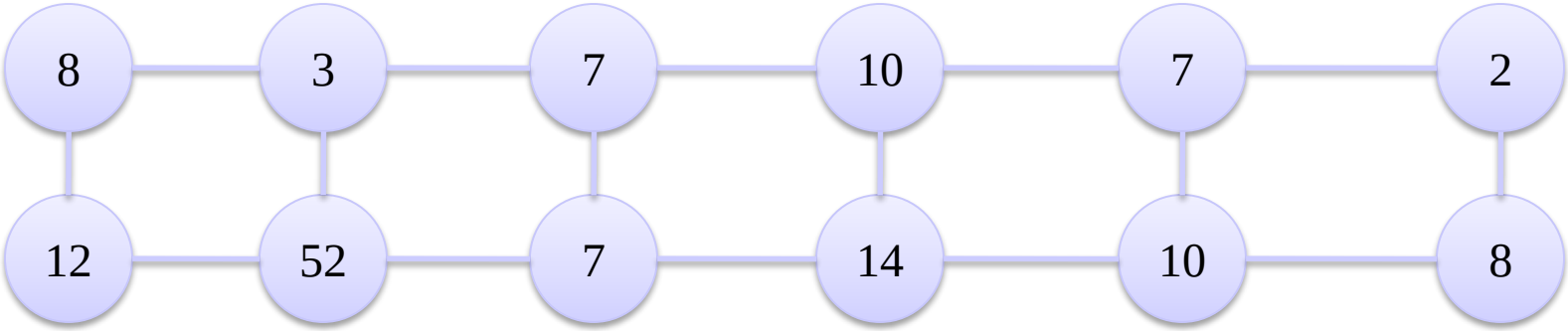
- Is there a simple greedy algorithm?

Possible greedy algorithms

- Repeatedly add heaviest compatible vertex
- Look at all possible maximum-**size** independent sets
- Counterexamples on previous slide.

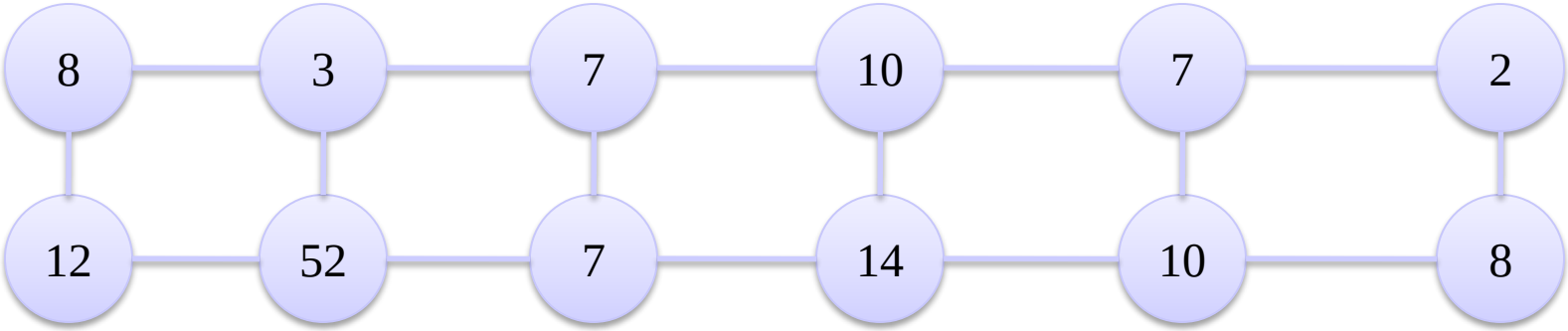
Exercise

- Do the same with a 2 x n grid graph



Exercise

- Do the same with a 2 x n grid graph



- Three types of subproblems:
  - grid(i)
  - gridTop(i)
  - gridBottom(i)