CSE 417 Algorithms

Lecture 20, Autumn 2020

Dynamic Programming

Announcements

Reading

- Dynamic Programming Examples:
  - 6.1-6.2, Weighted Interval Scheduling
  - 6.3 Segmented Least Squares
  - 6.4 Knapsack and Subset Sum
  - 6.6 String Alignment
    - 6.7* String Alignment in linear space
  - 6.8 Shortest Paths (again)
  - 6.9 Negative cost cycles
    - How to make an infinite amount of money

Dynamic Programming

- Key ideas
  - Construct optimization function
  - Express solution in terms of sub problems
  - Order sub problems to avoid recomputation
- Important detail
  - Record information to construct solution

Weighted Interval Scheduling

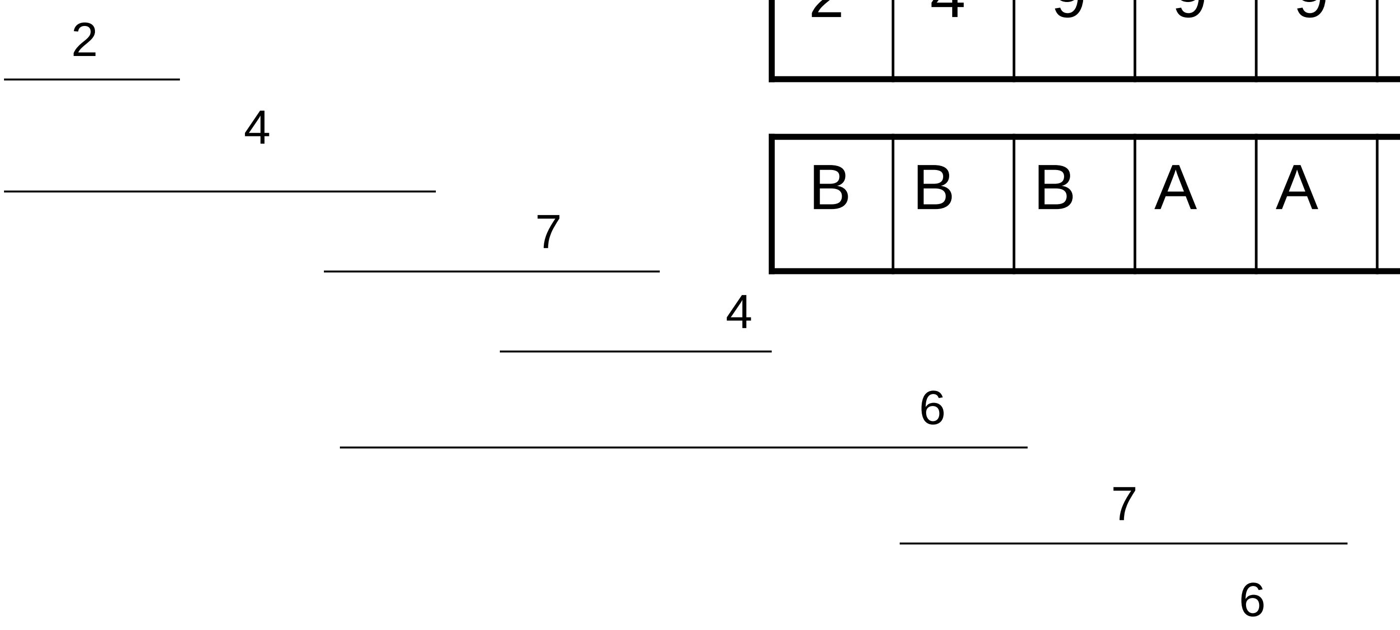
Opt[j] = max (Opt[j – 1], w_j + Opt[p[j]])

- Opt[j] : Maximum value solution from $I_1, I_2, \ldots I_j$
- Consider solutions not containing, and containing I_j

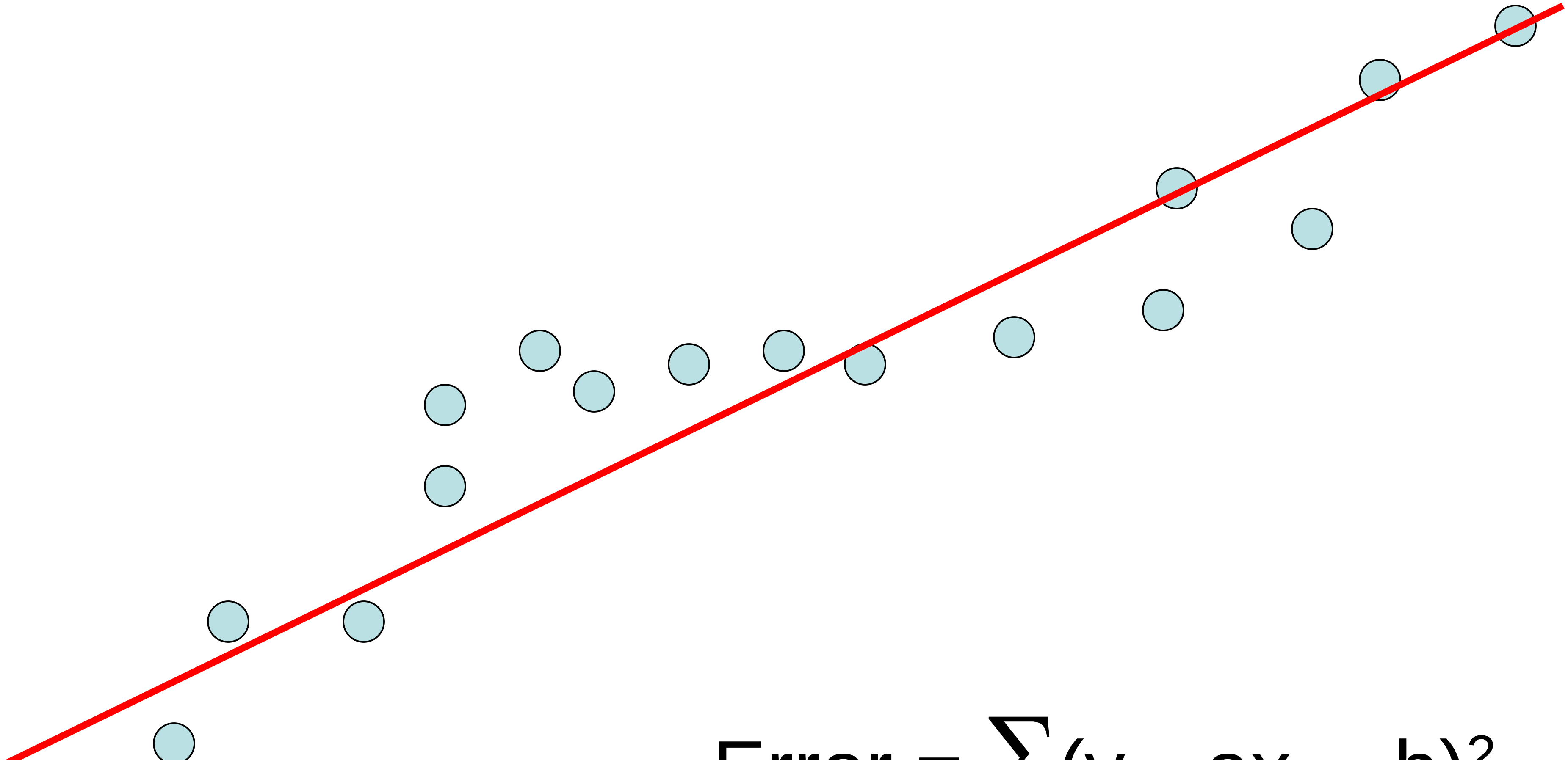
Record which case is used in Opt computation

2	4	9	9	9	16	16

B	B	B	A	A	B	A

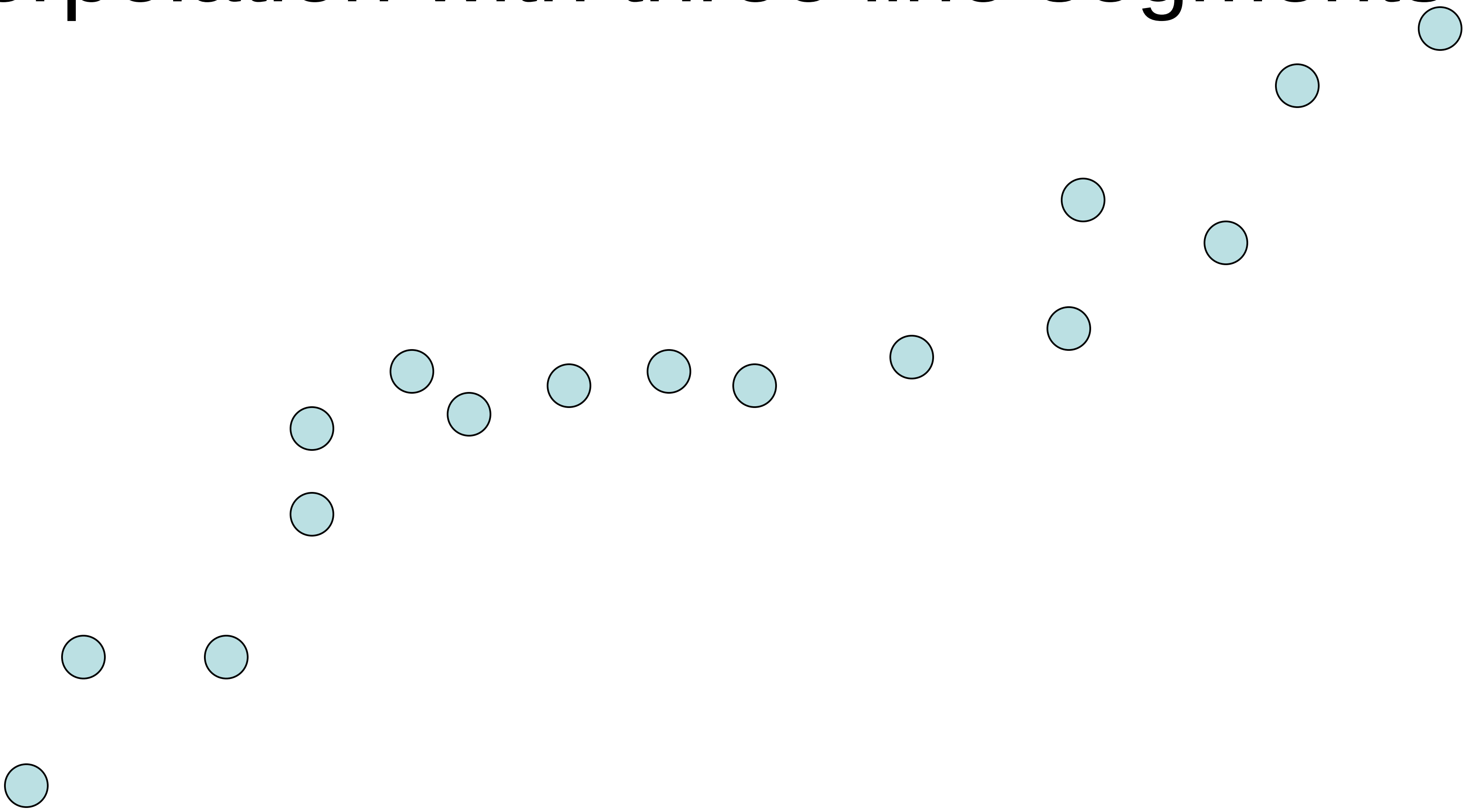


Optimal linear interpolation

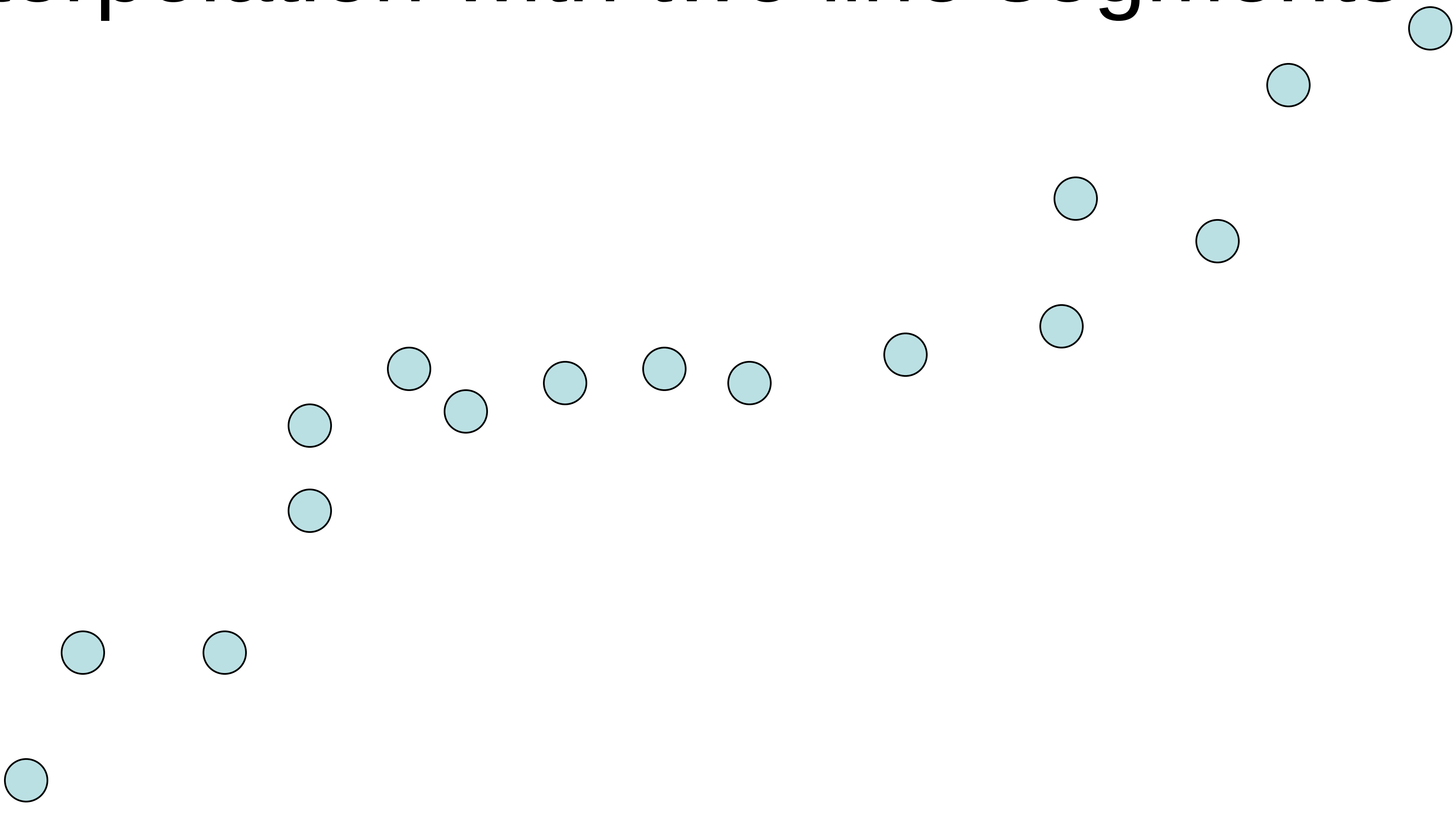


$$\text{Error} = \sum (y_i - ax_i - b)^2$$

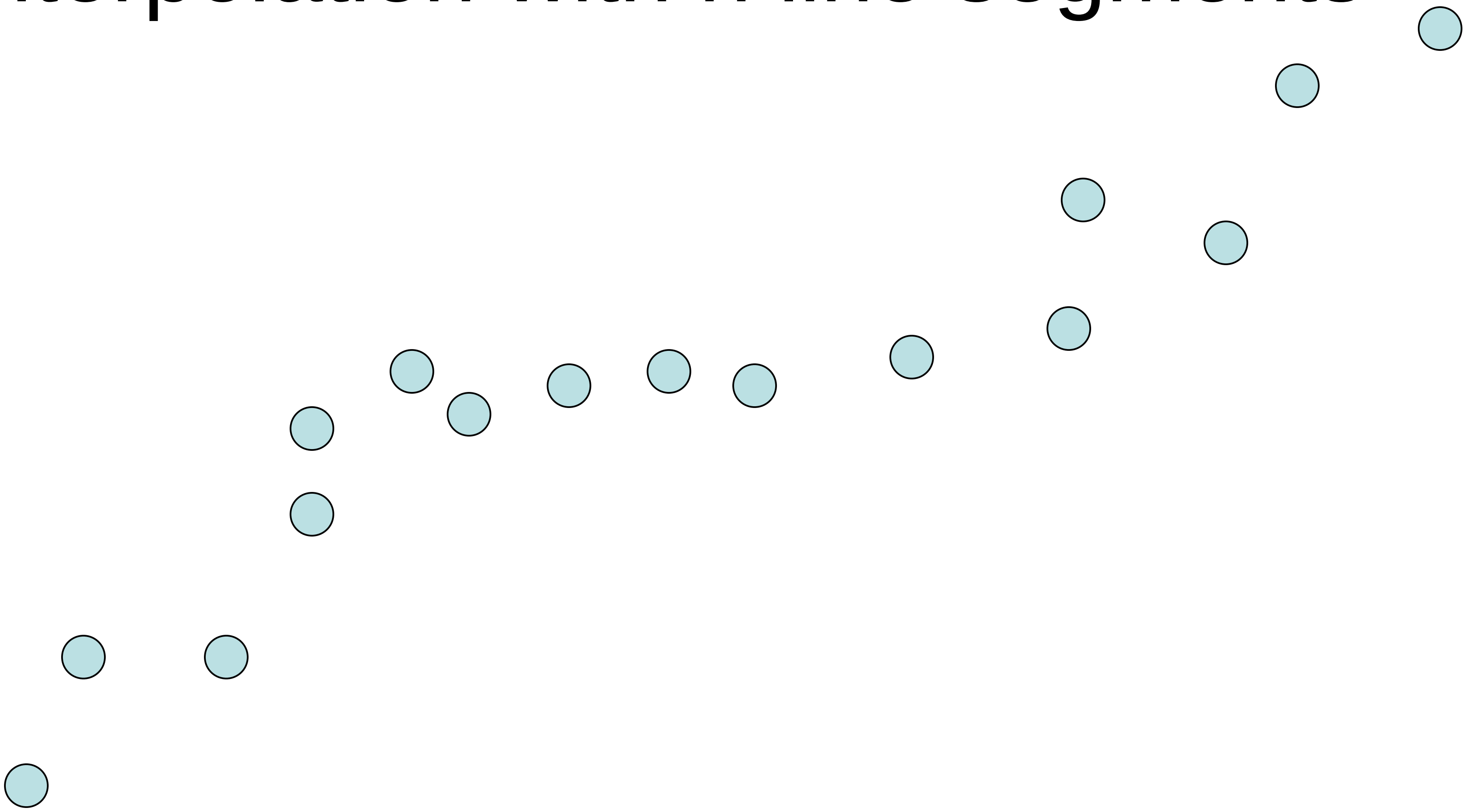
What is the optimal linear interpolation with three line segments



What is the optimal linear interpolation with two line segments

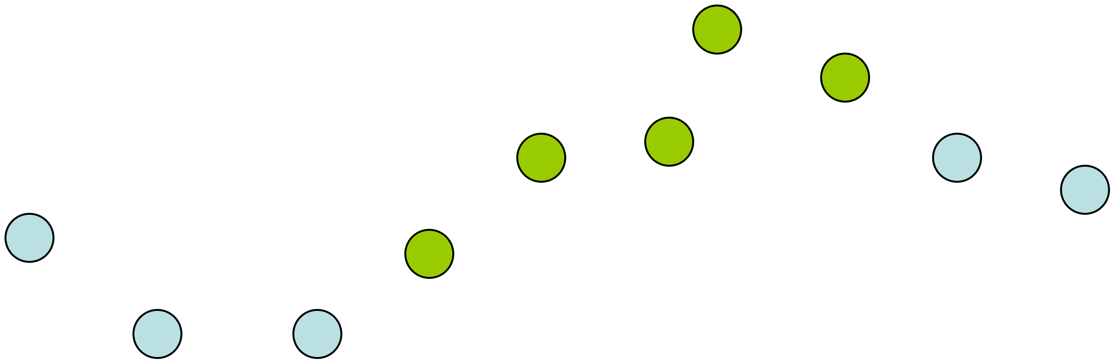


What is the optimal linear interpolation with n line segments



Notation

- Points p_1, p_2, . . ., p_n ordered by x-coordinate ($p_i = (x_i, y_i)$)
- $E_{i,j}$ is the least squares error for the optimal line interpolating p_i, . . . p_j



Optimal interpolation with two segments

- Give an equation for the optimal interpolation of $p_1,\ldots,p_n$ with two line segments

- $E_{i,j}$ is the least squares error for the optimal line interpolating $p_i, \ldots p_j$

Optimal interpolation with k segments

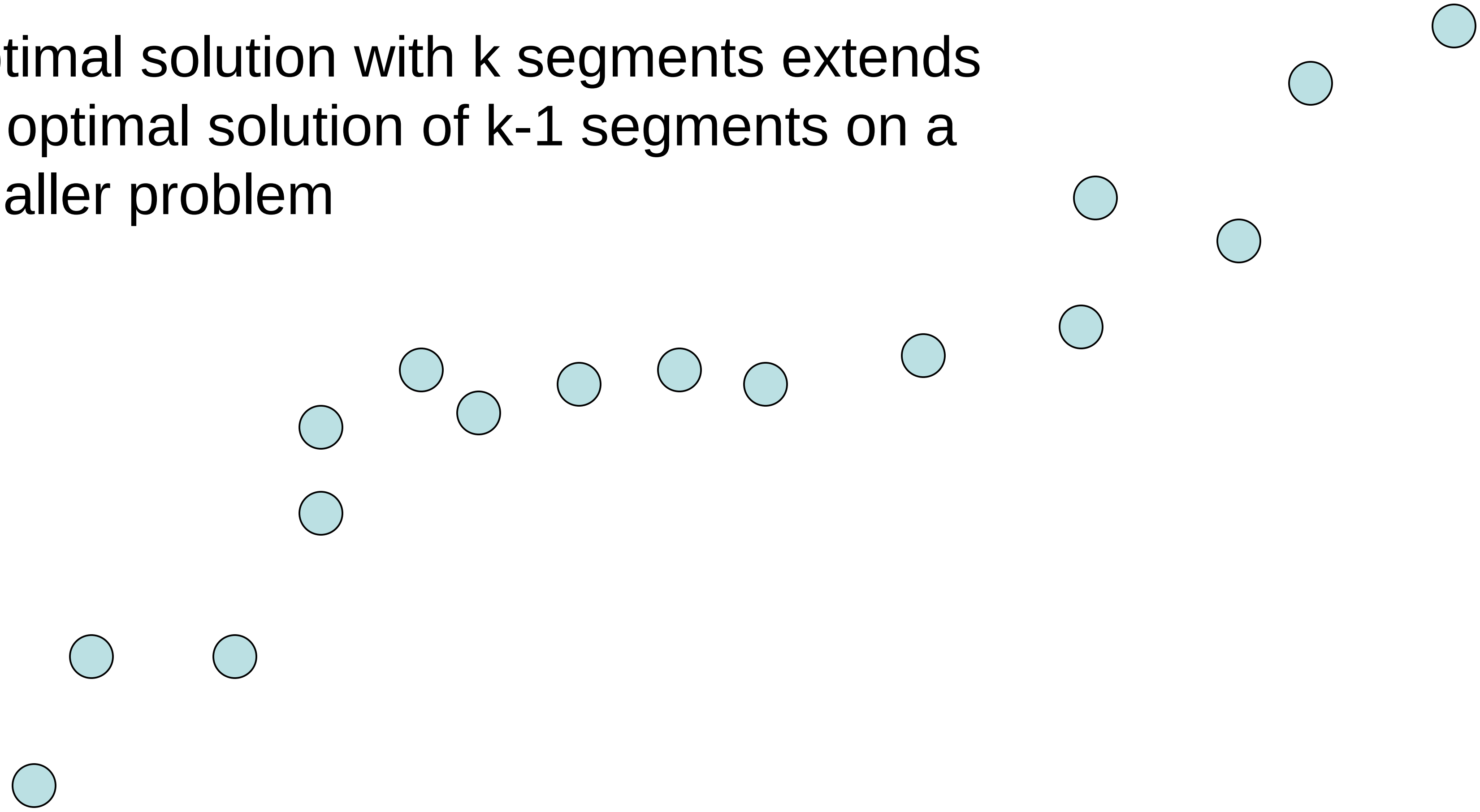
- Optimal segmentation with three segments
  - $\text{Min}_{i,j}\{E_{1,i} + E_{i,j} + E_{j,n}\}$
  - $O(n^2)$ combinations considered
- Generalization to k segments leads to considering $O(n^{k-1})$ combinations

$Opt_k[\ j\ ]$: Minimum error approximating $p_1 \ldots p_j$ with k segments

How do you express $Opt_k[\ j\ ]$ in terms of $Opt_{k-1}[1],\ldots,Opt_{k-1}[\ j\ ]$?

Optimal sub-solution property

Optimal solution with k segments extends an optimal solution of k-1 segments on a smaller problem



Optimal multi-segment interpolation

Compute Opt[k, j] for 0 < k < j < n

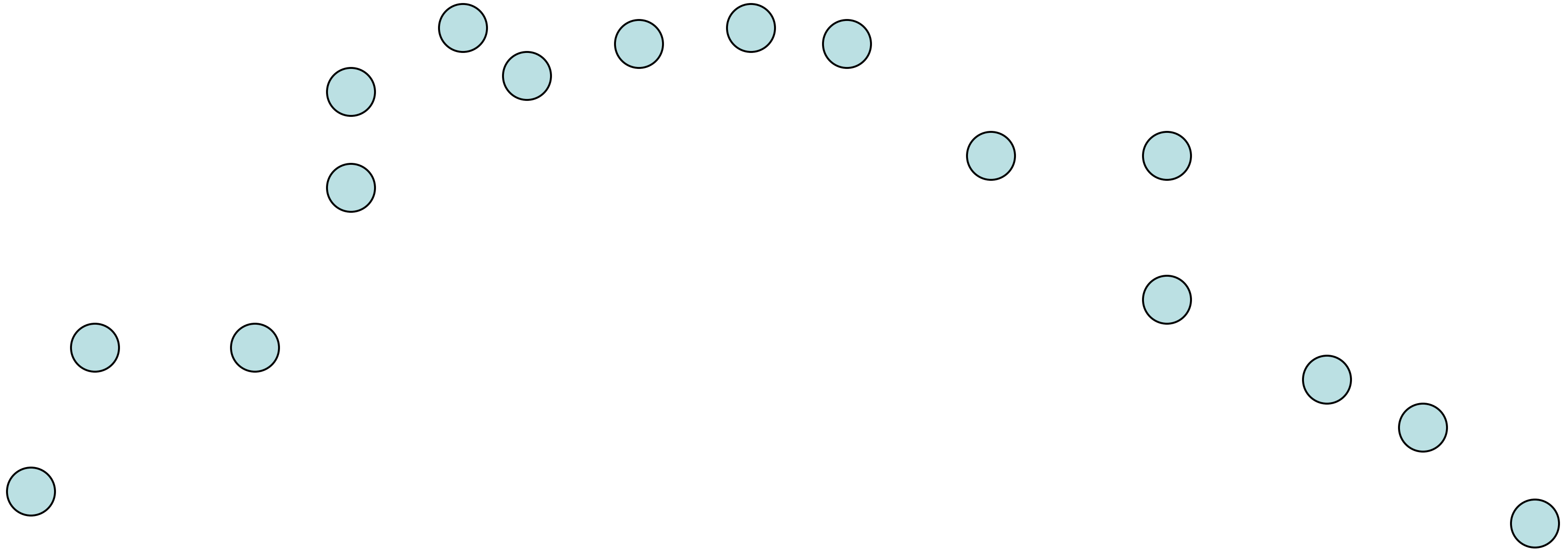
```
for j = 1 to n
    Opt[1, j] = E_{1,j};

for k = 2 to n-1
    for j = 2 to n
        t = E_{1,j}
        for i = 1 to j-1
            t = min(t, Opt[k-1, i] + E_{i,j})
        Opt[k, j] = t
```

Determining the solution

- When Opt[k, j] is computed, record the value of i that minimized the sum
- Store this value in an auxiliary array
- Use to reconstruct solution

Variable number of segments

- Segments not specified in advance
- Penalty function associated with segments
- Cost = Interpolation error + C x #Segments



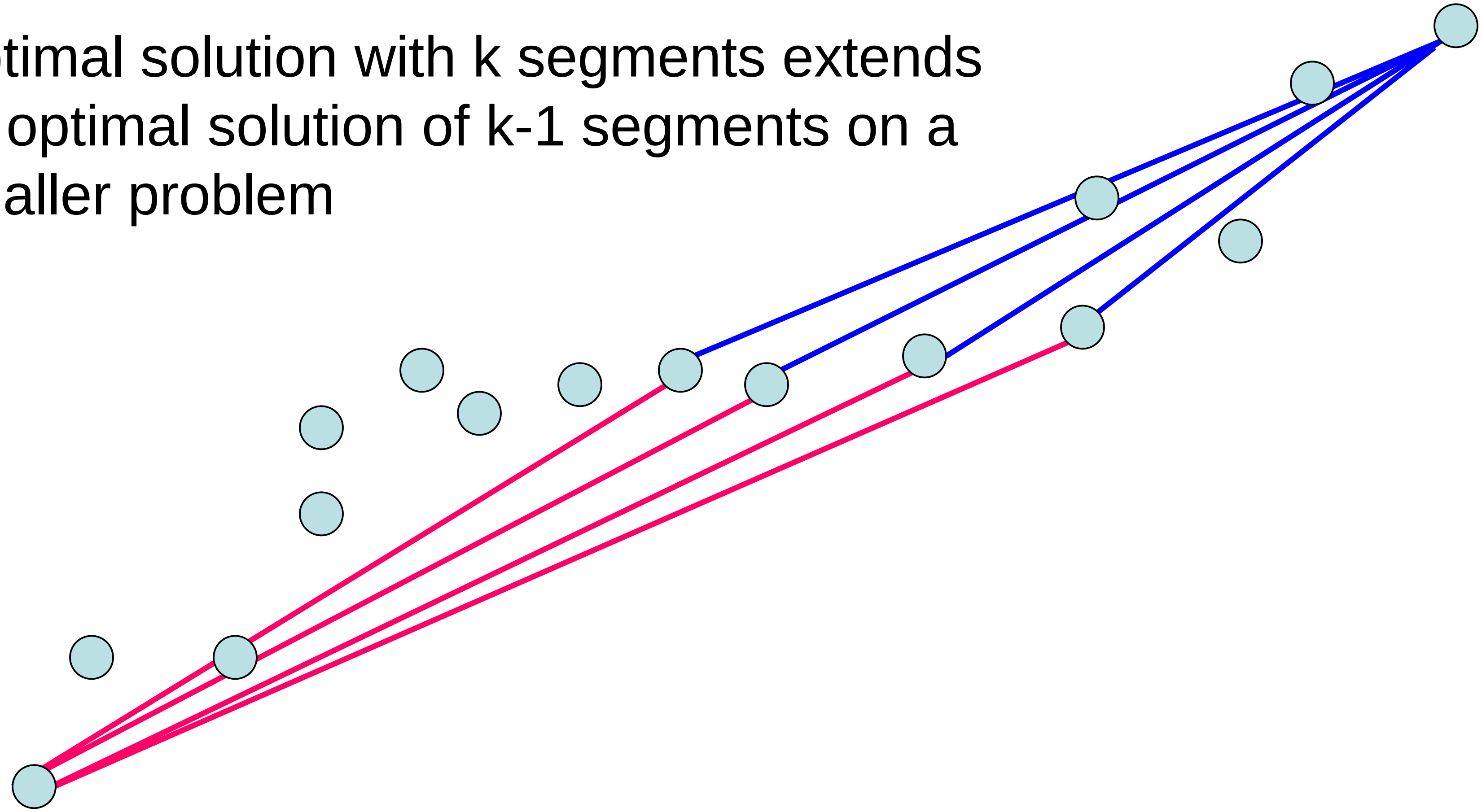
Penalty cost measure

- $Opt[\ j\ ] = \min(E_{1,j}, \min_i(Opt[\ i\ ] + E_{i,j} + P))$

Summary for Segmented Interpolation

$$Opt_k[\, j \,] = \min_i \{ Opt_{k-1}[\, i \,] + E_{i,j} \} \text{ for } 0 < i < j$$

Optimal solution with k segments extends an optimal solution of k-1 segments on a smaller problem



Subset Sum Problem

- Let $w_1,\ldots,w_n$ = {6, 8, 9, 11, 13, 16, 18, 24}
- Find a subset that has as large a sum as possible, without exceeding 50

Adding a variable for Weight

- Opt[j, K] the largest subset of $\{w_1, \ldots, w_j\}$ that sums to at most K
- {2, 4, 7, 10}
  - Opt[2, 7] =
  - Opt[3, 7] =
  - Opt[3,12] =
  - Opt[4,12] =

Subset Sum Recurrence

- Opt[j, K] the largest subset of $\{w_1, \ldots, w_j\}$ that sums to at most K

Subset Sum Grid

$Opt[\,j, K] = \max(Opt[\,j - 1, K], Opt[\,j - 1, K - w_j] + w_j)$

4																	
3																	
2																	
1																	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

{2, 4, 7, 10}

Subset Sum Code

```
for j = 1 to n
    for k = 1 to W
        Opt[j, k] = max(Opt[j-1, k], Opt[j-1, k-w_j] + w_j)
```

Knapsack Problem

- Items have weights and values
- The problem is to maximize total value subject to a bound on weght
- Items $\{I_1, I_2, \ldots I_n\}$
  - Weights $\{w_1, w_2, \ldots, w_n\}$
  - Values $\{v_1, v_2, \ldots, v_n\}$
  - Bound K
- Find set S of indices to:
  - Maximize $\sum_{i \varepsilon S} v_i$ such that $\sum_{i \varepsilon S} w_i <= K$

Knapsack Recurrence

Subset Sum Recurrence:

$$Opt[\ j, K] = \max(Opt[\ j - 1, K], Opt[\ j - 1, K - w_j] + w_j)$$

Knapsack Recurrence:

Knapsack Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + v_j)

4																	
3																	
2																	
1																	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Weights {2, 4, 7, 10} Values: {3, 5, 9, 16}

Alternate approach for Subset Sum

- Alternate formulation of Subset Sum dynamic programming algorithm
  - Sum[i, K] = true if there is a subset of $\{w_1,\ldots w_i\}$ that sums to exactly K, false otherwise
  - Sum [i, K] = Sum [i -1, K] **OR** Sum[i - 1, K - w_i]
  - Sum [0, 0] = true; Sum[i, 0] = false for i != 0

- To allow for negative numbers, we need to fill in the array between K_{min} and K_{max}

Run time for Subset Sum

- With n items and target sum K, the run time is O(nK)
- If K is 1,000,000,000,000,000,000,000,000 this is very slow
- Alternate brute force algorithm: examine all subsets: $O(n2^n)$