CS 7800: Advanced Algorithms

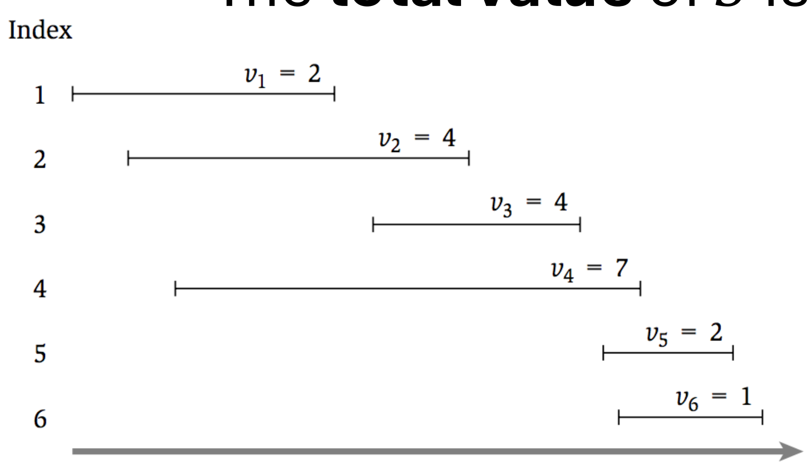
Class 5: Dynamic Programming II

- Finish Weighted Interval Scheduling
- Segmented Least Squares

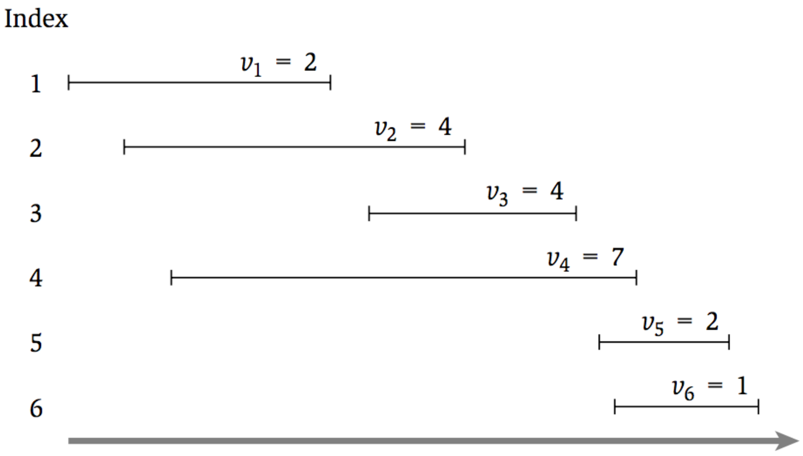
Jonathan Ullman
September 19, 2025

Weighted Interval Scheduling

- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S **maximizing** the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \ldots, n\}$
  - A schedule S is **compatible** if no $i, j \in S$ overlap
  - The **total value** of S is $\sum_{i \in S} v_i$

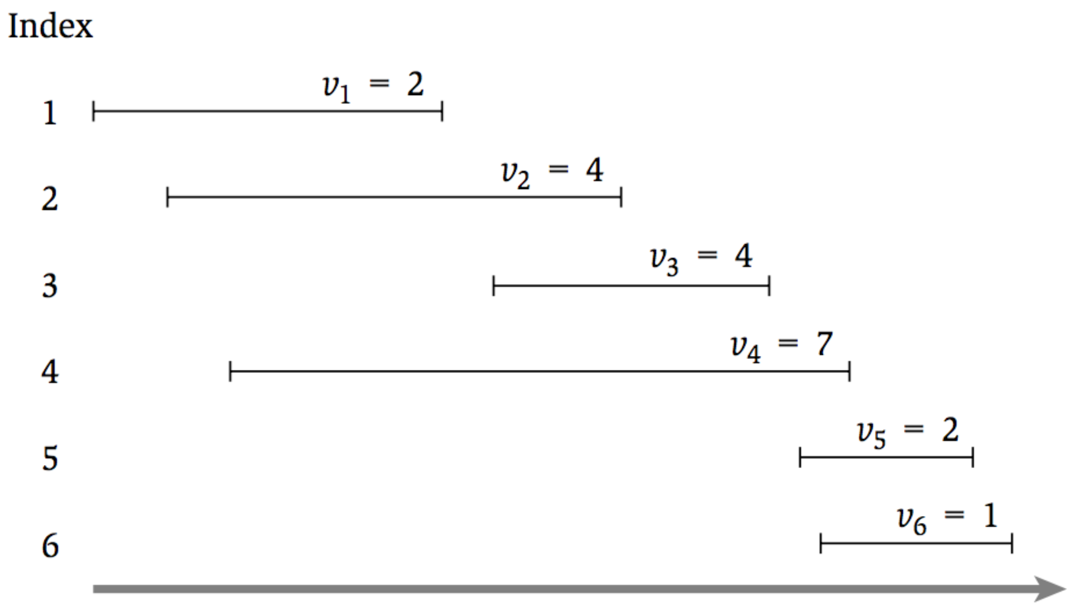


Finding the Recurrence



Finding the Optimal Solution

But we want a schedule, not a value!



M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

Weighted Interval Scheduling Recap

- There is an $O(n \log n)$ algorithm for the weighted interval scheduling problem
  - Generalizes the greedy alg for the unweighted version
  - Our first example of dynamic programming

- **Dynamic Programming Recipe:**

  (1) identify a set of **subproblems**

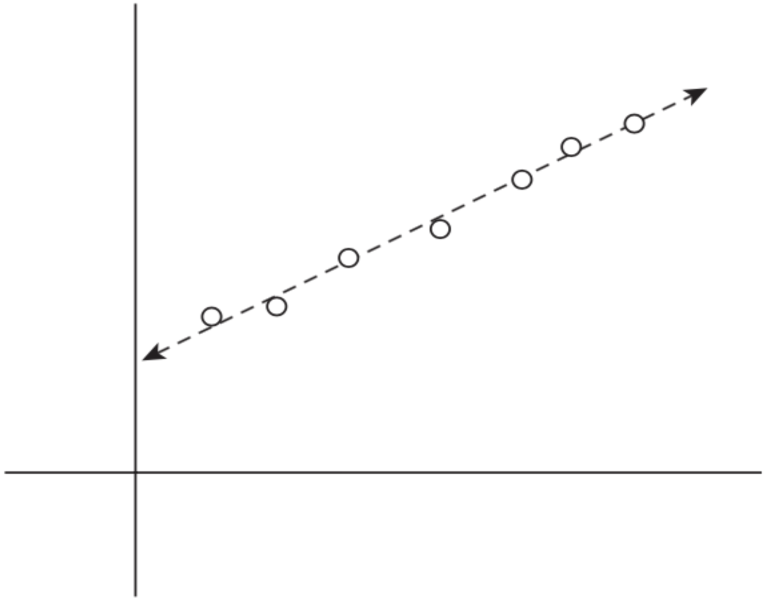
  (2) relate the subproblems via a **recurrence**

  (3) design an algorithm to **efficiently solve** the recurrence

  (4) recover the **actual solution** at the end

(Ordinary) Least Squares

- **Input:** n data points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
- **Output:** the line L (i.e. $y = ax + b$) that fits **best**
  - **best** = minimizes $error(L, P) = \sum_i (y_i - ax_i - b)^2$



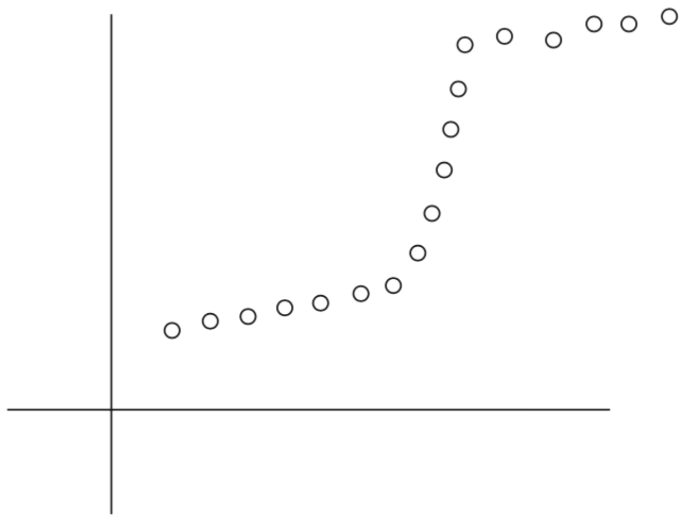
$$a = \frac{n\sum x_i y_i - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - (\sum x_i)^2}$$

$$b = \frac{\sum y_i - a\sum x_i}{n}$$

- There is an $O(n)$ time algorithm for finding the line of best fit

Segmented Least Squares

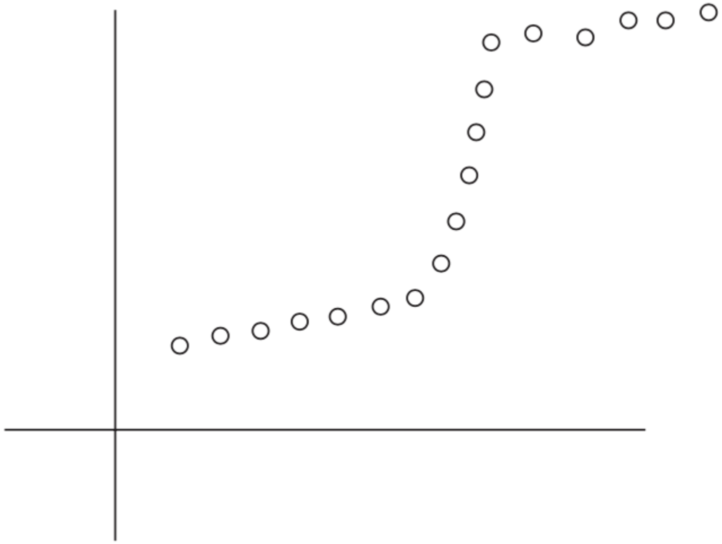
- **Input:** n data points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
- What if the data does not look like a line?



- Some data can be described better by more than one line **segment**
- But, using $\geq {}^{n}/_{2}$ segments defeats the purpose!

Segmented Least Squares

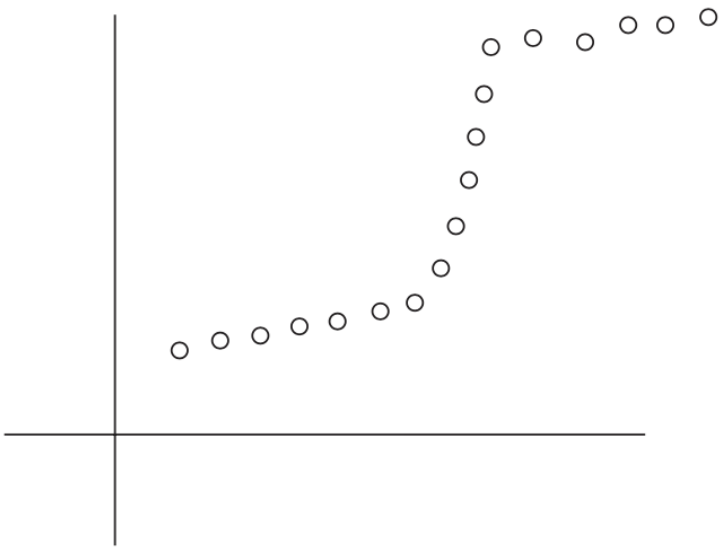
- **Input:** n data points $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$, **cost parameter** $C > 0$
  - Assume $x_1 < x_2 < \cdots < x_n$
- **Output:** a partition of P into contiguous (disjoint) segments $S_1, S_2, \dots, S_m$, lines $L_1, L_2, \dots, L_m$, minimizing total "cost"



cost($S_1, \dots, S_m, L_1, \dots, L_m$)
= $mC + \sum_{i=1}^{m} error(L_i, S_i)$

Segmented Least Squares

- **First observation:** for every segment S_j, L_j must be the (single) line of best fit for S_j
  - Let $L^*_{i,j}$ be the optimal line for $\{p_i, \ldots, p_j\}$
  - Let $\varepsilon_{i,j} = error\left(L^*_{i,j}, \{p_i, \ldots, p_j\}\right)$



Can compute $\varepsilon_{i,j}$ for all i, j in $O(n^3)$ time straightforwardly,

…or $O(n^2)$ time with more cleverness

Writing the Recurrence

Let $L_{i,j}^*$ be the optimal line for $\{p_i, \ldots, p_j\}$

Let $\varepsilon_{i,j} = error(L_{i,j}^*, \{p_i, \ldots, p_j\})$

SLS: Take I

```
// All inputs are global vars
FindOPT(n):
  if (n = 0): return 0
  elseif (n = 1,2): return C
  else:
    return min_{1≤i≤n}(ε_{i,n} + C + FindOPT(i − 1))
```

$$\min_{1 \le i \le n} (\varepsilon_{i,n} + C + \text{FindOPT}(i-1))$$

Runtime:

SLS: Take III ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n):
  M[0] ← 0, M[1] ← C, M[2] ← C
  for (j = 3,…,n):
    M[j] ← min_{1≤i≤j}( ε_{i,j} + C + M[i − 1] )
  return M[n]
```

$$M[j] \leftarrow \min_{1 \le i \le j} \left(\varepsilon_{i,j} + C + M[i-1] \right)$$

Runtime:

Finding Segments

Let $L_{i,j}^*$ be the optimal line for $\{p_i, \ldots, p_j\}$

Let $\varepsilon_{i,j} = error\left(L_{i,j}^*, \{p_i, \ldots, p_j\}\right)$

Finding Segments

```
// All inputs are global vars
// M[0:n] contains solutions to subproblems
FindSol(M,n):
  if (n = 0): return ∅
  elseif (n = 1): return {1}
  elseif (n = 2): return {1,2}
  else:
    Let x ← argmin_{1≤i≤n} (ε_{i,n} + C + M[i − 1]):
    return {x,…,n} + FindSol(M,x-1)
```

Runtime:

Weighted Interval Scheduling Recap

- There is an $O(n^2)$ time algorithm for the weighted interval scheduling problem
  - Second example of dynamic programming
  - Canonical example of partitioning a line into segments

- **Dynamic Programming Recipe:**

  (1) identify a set of **subproblems**

  (2) relate the subproblems via a **recurrence**

  (3) design an algorithm to **efficiently solve** the recurrence

  (4) recover the **actual solution** at the end