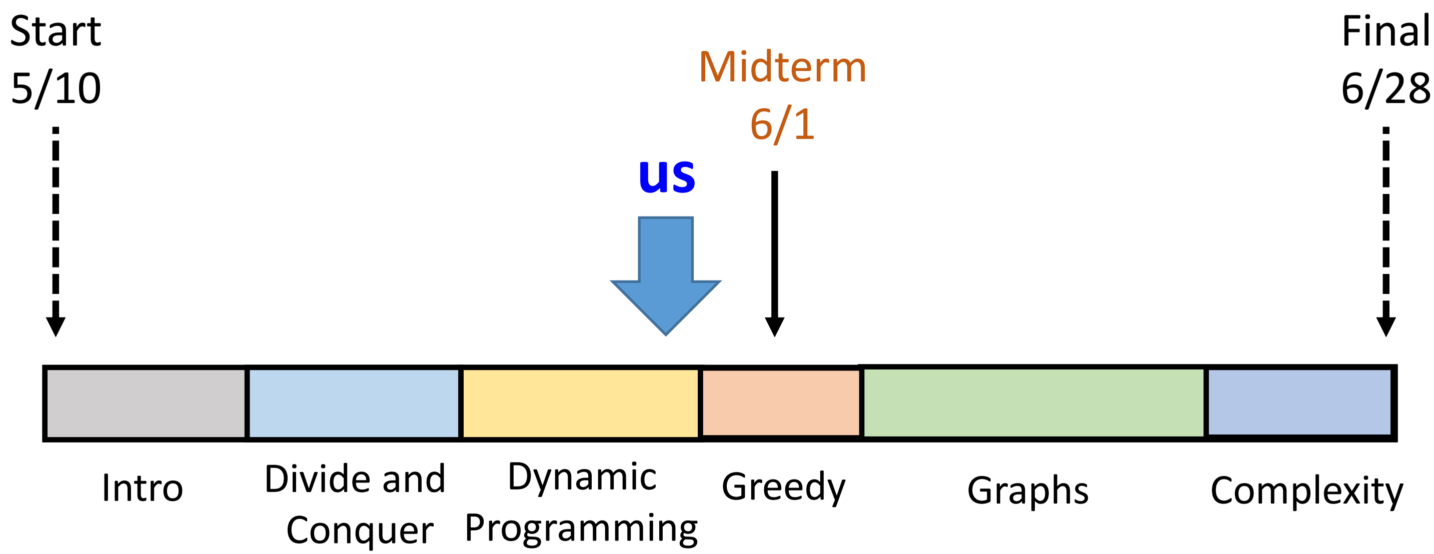
CS3000: Algorithms & Data
Drew van der Poel

Lecture 10

- Dynamic Programming: Knapsack (Finish)
- Dynamic Programming: Segmented Least Squares

May 25, 2021

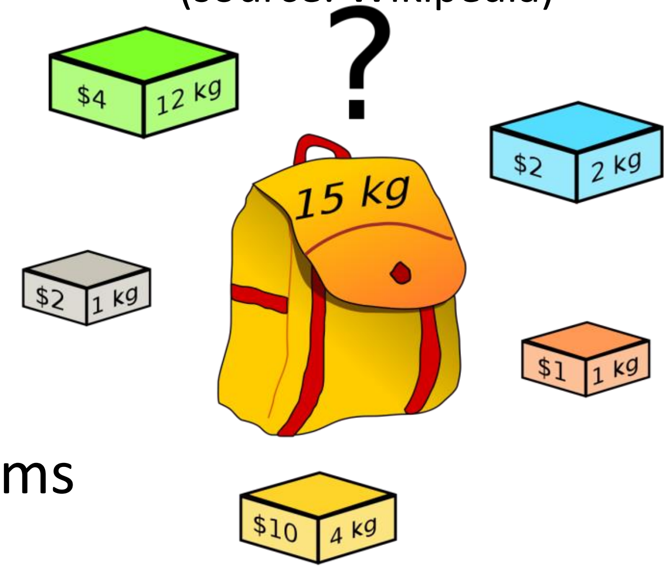
Outline



Last class: dynamic programming: Knapsack

Next class: greedy algorithms: Scheduling

(source: Wikipedia)



The Knapsack Problem

- **Input:** n items for your knapsack
  - value v_i and a weight $w_i \in \mathbb{N}$ for n items
  - capacity of your knapsack $T \in \mathbb{N}$
- **Output:** the most valuable subset of items that fits in the knapsack
  - Subset $S \subseteq \{1, \dots, n\}$
  - Value $V_S = \sum_{i \in S} v_i$ as large as possible
  - Weight $W_S = \sum_{i \in S} w_i$ at most T

$n =$ $T =$

$v_1 =$ $w_1 =$

$v_2 =$ $w_2 =$

$v_3 =$ $w_3 =$

$v_4 =$ $w_4 =$

$v_5 =$ $w_5 =$

- **Want:** $\mathbf{argmax}_{S \subseteq \{1,\dots,n\}} V_S$ s.t. $W_S \leq T$

- **(SubsetSum:** $v_i = w_i$,
- **TugOfWar:** $v_i = w_i, T = \frac{1}{2} \sum_i v_i$)

Knapsack - recurrence

(1)

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - $OPT(j, S) = OPT(j-1, S)$
- **Case 2:** $j \in O_{j,S}$
  - $OPT(j, S) = v_j + OPT(j-1, S - w_j)$

(2)

Recurrence:

$$\mathrm{OPT}(j, S) = \begin{cases} \max\{OPT(j-1, S), v_j + OPT(j-1, S-w_j)\} & S \geq w_j \\ OPT(j-1, S) & S < w_j \end{cases}$$

Base Cases:

$$\mathrm{OPT}(j, 0) = \mathrm{OPT}(0, S) = 0$$

Knapsack ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n,T):
  M[0,S] ← 0, M[j,0] ← 0

  for (j = 1,…,n):
    for (S = 1,…,T):
      if (wj > S): M[j,S] ← M[j-1,S]
      else: M[j,S] ← max{M[j-1,S],vj + M[j-1,S-wj]}

  return M[n,T]
```

O(1) each

runtime: O(nT)

nT iterations → # of loops

each loop → O(1)

Ask the Audience

Space: $O(nT)$ - $(T+1) \times (n+1)$

entries in DP table

dominates

- Input: $T = 8, n = 3$
  - $w_1 = 2$, $v_1 = 4$
  - $w_2 = 3$, $v_2 = 5$
  - $w_3 = 5$, $v_3 = 8$

items (j)

3	0	0	4	5	5	9	9	12	$\boxed{13}$
2	0	0	4	5	5	9	9	9	9
1	0	0	$\boxed{4}$	4	4	4	4	4	4
0	0	0	0	0	0	0	0	0	0
-	0	1	2	3	4	5	6	7	8

capacities (S)

$$\mathrm{OPT}(j,S) = \begin{cases} \max\{OPT(j-1,S), v_j + OPT(j-1,S-w_j)\} & \text{If } S \geq w_j \\ \rightarrow OPT(j-1,S) & \text{If } S < w_j \end{cases}$$

Filling the Knapsack

- Let $\boldsymbol{O_{j,S}}$ be the **optimal subset of items** $\{1, \dots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - Use opt. solution for items 1 to j-1 in a knapsack of size S
- **Case 2:** $j \in O_{j,S}$
  - Use j + opt. solution for items 1 to j-1 in a knapsack of size $S - w_j$

add j to subset

Filling the Knapsack

```
// All inputs are global vars
// M[0:n,0:T] contains solutions to subproblems
FindSol(M,n,T):
  if (n = 0 or T = 0): return ∅
  else:
    if (w_n > T): return FindSol(M,n-1,T)
    else:
      if (M[n-1,T] > v_n + M[n-1,T-w_n]):
        return FindSol(M,n-1,T)
      else:
        return {n} + FindSol(M,n-1,T-w_n)
```

R/T: $O(n)$ each call decrements n

Knapsack Wrapup

- Can solve knapsack problems in time/space $O(nT)$
  - **Recipe:**

    $OPT(j, S^*)$

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

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

    (3) find an **efficient implementation** of the recurrence (top down or bottom up)

    (4) **reconstruct the solution** from the DP table

DP Practice

Problem 2. *Dynamic Programming*

The dark lord Sauron loves to destroy the kingdoms of Middle Earth. But he just can't catch a break, and is always eventually defeated. After a defeat, he requires three epochs to rebuild his strength and once again rise to destroy the kingdoms of Middle Earth. In this problem, you will help Sauron decide in which epochs to rise and destroy the kingdoms of Middle Earth.

The input to the algorithm consists of the numbers $x_1, \ldots, x_n$ representing the number of kingdoms in each epoch. If Sauron rises in epoch i then he will destroy all x_i kingdoms, but will not be able to rise again during epochs $i+1, i+2$, or $i+3$. We call a set $S \subseteq \{1, \ldots, n\}$ of epochs *valid* if it satisfies this constraint that $|i-j| \geq 4$ for all $i, j \in S$, and its *value* is $\sum_{i \in S} x_i$. You will design an algorithm that outputs a valid set of epochs with the maximum possible value.

Example: Suppose there are $(1, 7, 8, 2, 6, 3)$ kingdoms of Middle Earth in epochs $1, \ldots, 6$. Then the optimal set of epochs for Sauron to rise up and destroy the kingdoms of Middle Earth is $S = \{2, 6\}$, during which he destroys 10 kingdoms, 7 in the 2nd epoch and 3 in the 6th epoch.

Using DP…

*** describe the set of subproblems you consider**

*** give a recurrence expressing the solution to each subproblem in terms of the solution to smaller subproblems**

***sketch pseudocode of your algorithm & give the runtime**

***describe how you would recover the solution (epochs) if asked**

Segmented Least Squares

Dynamic Programming Recap

- **Recipe:**

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

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

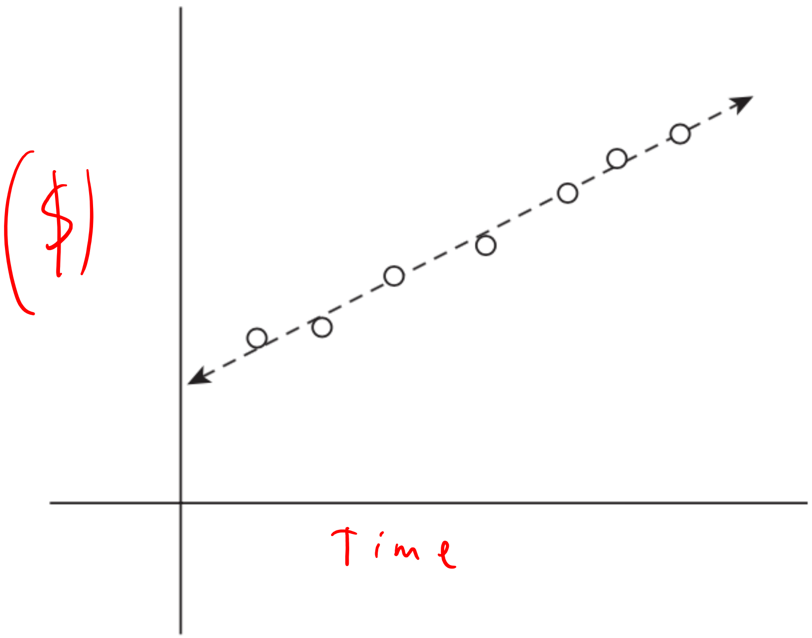
  (3) find an **efficient implementation** of the recurrence (top down or bottom up)

  (4) **reconstruct the solution** from the DP table

* TODAY: multi-way case analysis

Background: 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 $\boxed{error(L, P) = \sum_i (y_i - ax_i - b)^2}$



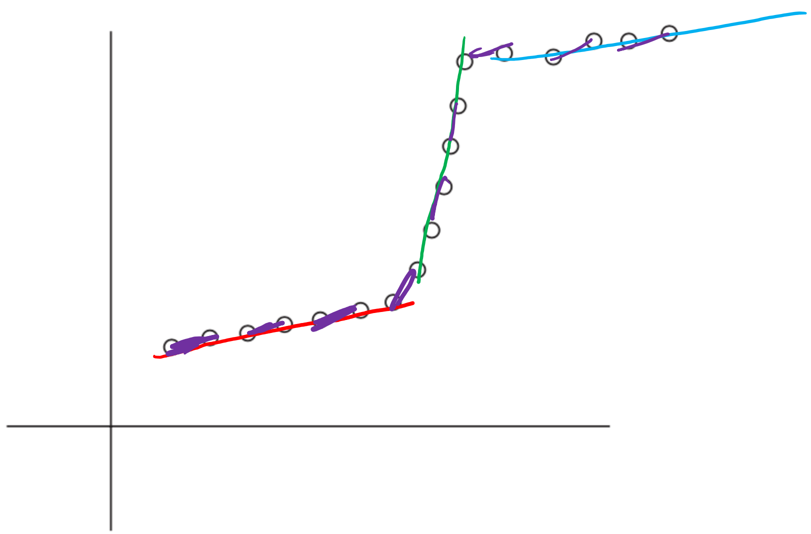
$$a = \frac{n\sum x_i y_i - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - (\sum x_i)^2}$$

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

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

Segmented Least Squares

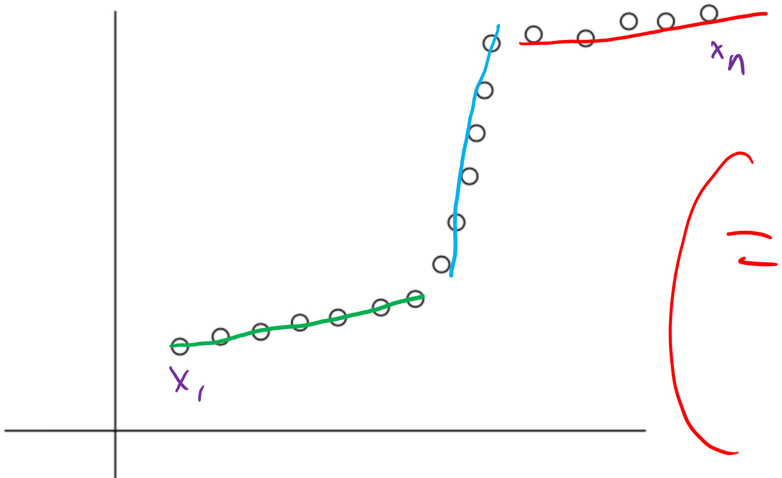
- **Input:** n data points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
- What if the data does not look like a line?



- Some data can be described better by > *1* line – call each group a *segment*
- Using *n/2* segments defeats the purpose (**how to prevent this?**)

Segmented Least Squares

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



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

$\left(= 3C + error(L_1, S_1) + error(L_2, S_2) + error(L_3, S_3) \right)$

- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, weighted interval scheduling, **segmented least squares**

- Alg. techniques: divide & conquer, dynamic programming

- Analysis: asymptotic analysis, recursion trees, Master Thm.

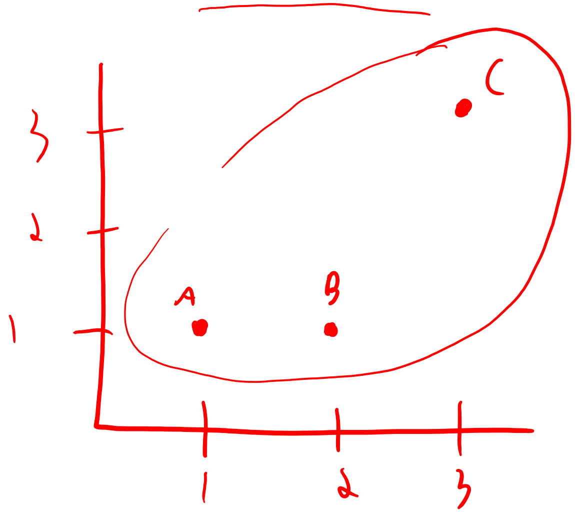
- Proof techniques: (strong) induction, contradiction

SLS Example

n=3

- **Input:** {A=(1,1), B=(2,1), C=(3,3)}

Potential segment	Optimal line	Error
[A]	y = 1	0
[B]	y = 1	0
[C]	y = 3	0
[A,B]	y = 1	0
[B,C]	y = 2x-3	0
[A,B,C]	y = x – 1/3	2/3



Possible Solutions

1. ~~{[A], [B], [C]}~~ 3C
2. {[A], [B,C]} = {[A,B], [C]} 2C
3. {[A,B,C]}

SLS Example

- **Input:** {A=(1,1), B=(2,1), C=(3,3)}

Potential segment (S)	Optimal line (L)	Error
[A]	y = 1	0
[B]	y = 1	0
[C]	y = 3	0
[A,B]	y = 1	0
[B,C]	y = 2x-3	0
[A,B,C]	y = x – 1/3	2/3

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

m=2 **Partition 1:** [A], [B,C]

m=1 **Partition 2:** [A,B,C]

$2C + 0 + 0 = 2C$

$1C + 2/3 = C + 2/3$

$2C$ vs. $C + 2/3$

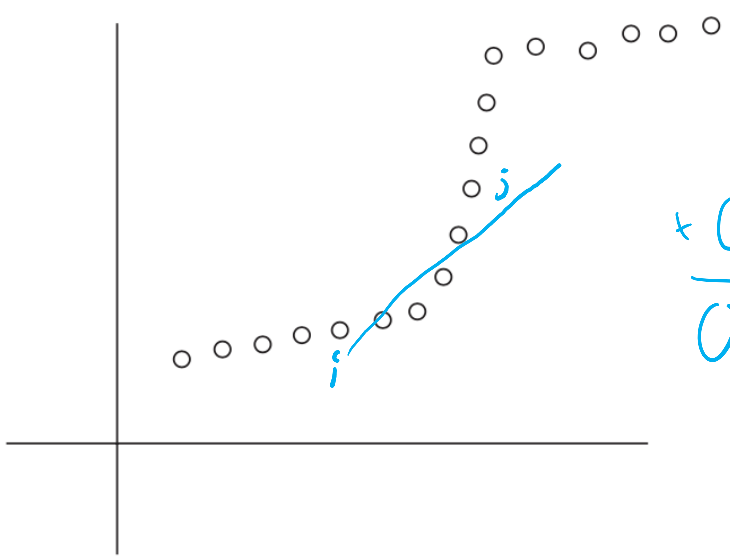
if $C > 2/3 \rightarrow$ P2 is opt

else $C < 2/3 \rightarrow$ P1 is opt

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

PRE-PROCESSING



$O(n^2)$ pairs

$+ \; O(n)$ best fit

$O(n^3)$

Can compute $\varepsilon_{i,j}$ for all *(i, j)* in $\boldsymbol{O(n^3)}$ time straightforwardly, or $\boxed{\boldsymbol{O(n^2)}}$ time with more cleverness

SLS

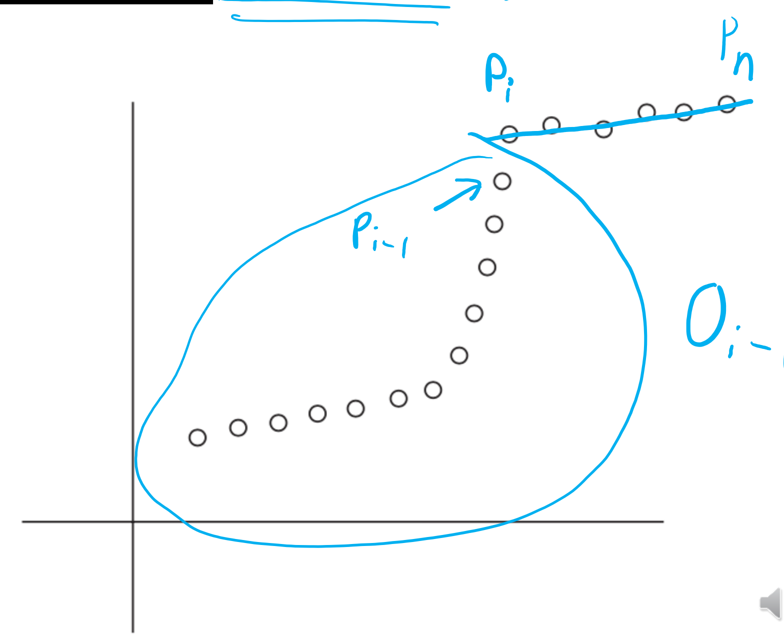
Let $L^*_{i,j}$ be the optimal line for $\{p_i, \dots, p_j\}$
Let $\varepsilon_{i,j} = error\left(L^*_{i,j}, \{p_i, \dots, p_j\}\right)$

- Let O_j be the **optimal** solution for $\{p_1, \dots, p_j\}$
- What is the final segment in O_j?

$[p_i, \dots\ p_j]$

If the final segment is $[p_i, \dots, p_n]$ $(1 \le i \le n)$
then the optimal solution
for $\{p_1, \dots, p_n\}$ is

$O_n = [p_i, \dots, p_n] \cup O_{i-1}$



Multi-way Choices

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

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

Value

$0 \le j \le n$

- Let $\mathrm{OPT}(j)$ be the **value** of the optimal solution for points $\{p_1, \dots, p_j\}$
- *Case **i***: final segment is $\{p_i, \dots, p_j\}$
  - optimal solution is $L_{i,j}^* \cup$ optimal sol. for $\{p_1, \dots, p_{i-1}\}$
  - can use any $i \in \{1, \dots, j\}$ (*O(j)* cases) $\Leftarrow$ multi-way cases
- Total cost is 1 + 2 + 3: $\varepsilon_{i,j} + C + OPT(i-1)$
  - 1. $\varepsilon_{i,j}$
  - 2. C
  - 3. $OPT(i-1)$

Multi-way Choices

Let $L^*_{i,j}$ be the optimal line for $\{p_i, \dots, p_j\}$
Let $\varepsilon_{i,j} = error\left(L^*_{i,j}, \{p_i, \dots, p_j\}\right)$

- Let $\mathrm{OPT}(j)$ be the **value** of the optimal solution for points $\{p_1, \dots, p_j\}$
- **Case i:** final segment is $\{p_i, \dots, p_j\}$
  - optimal solution is $L^*_{i,j}$ ∪ optimal sol. for $\{p_1, \dots, p_{i-1}\}$
  - can use any $i \in \{1, \dots, j\}$

j cases

Recurrence: $\mathrm{OPT}(j) = \min_{1 \le i \le j} \left(\underset{1}{\varepsilon_{i,j}} + \underset{2}{C} + \underset{3}{\mathrm{OPT}(i-1)} \right)$

Base cases:
$\mathrm{OPT}(0) = 0$
$\mathrm{OPT}(1) = \mathrm{OPT}(2) = C$

	$\varepsilon_{i,j} + C + \mathrm{OPT}(i-1)$
$i = 4$	70
$i = 5$	65

SLS: Take I

Not DP - no memoization

```
// All inputs are global vars
FindOPT(n):
  if (n = 0): return 0
  elseif (n = 1,2): return C
  else:
    return
```
$$\min_{1\le i\le n}(\varepsilon_{i,n}+C+\mathbf{FindOPT}(i-1))$$

Runtime: exponential - time :(

SLS: Take II ("Top-Down")

```
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← C, M[2] ← C
FindOPT(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← min_{1≤i≤n} (ε_{i,n} + C + FindOPT(i − 1))
    return M[n]
```

Have to fill $n-2$ elements $\left(\sum_{j=3}^{n} j = 3 + \ldots + n\right)$

$\hookrightarrow O(n)$

To fill *M[j]* we make j rec. calls

Total # of rec. calls: $O(n^2)$ **Total runtime:** $O(n^2)$

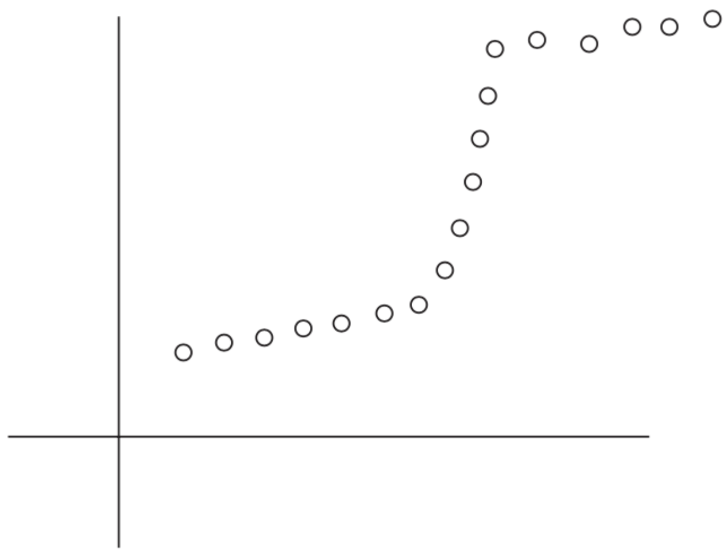
SLS: Take III ("Bottom-Up")

```
// 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]
```

$$\sum_{j=3}^{n} j = 3 + 4 + 5 + \dots + n$$

Runtime: $O(n^2)$

SLS: Take III ("Bottom-Up")



M[j]

M[0]	M[1]	M[2]	…	~~M[i]~~	…	M[n]
			…		…	

Finding Segments

Let $L^*_{i,j}$ be the optimal line for $\{p_i, \dots, p_j\}$

Let $\varepsilon_{i,j} = error\left(L^*_{i,j}, \{p_i, \dots, p_j\}\right)$

- Let O_j be the **optimal** solution for $\{p_1, \dots, p_j\}$
- Let $\mathrm{OPT}(j)$ be the **value** of the optimal solution for points $\{p_1, \dots, p_j\}$
- **Case i:** final segment is $\{p_i, \dots, p_j\}$
  - optimal solution is $L^*_{i,j} \cup$ optimal sol. for $\{p_1, \dots, p_{i-1}\}$
  - can use any $i \in \{1, \dots, j\}$

return i corresponding to the min

If x $== \mathbf{argmin}_{1 \le i \le n} \left(\varepsilon_{i,n} + C + M[i-1]\right)$

then $[p_x, \dots, p_n] \cup O_{x-1}$ is the solution!

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)
```

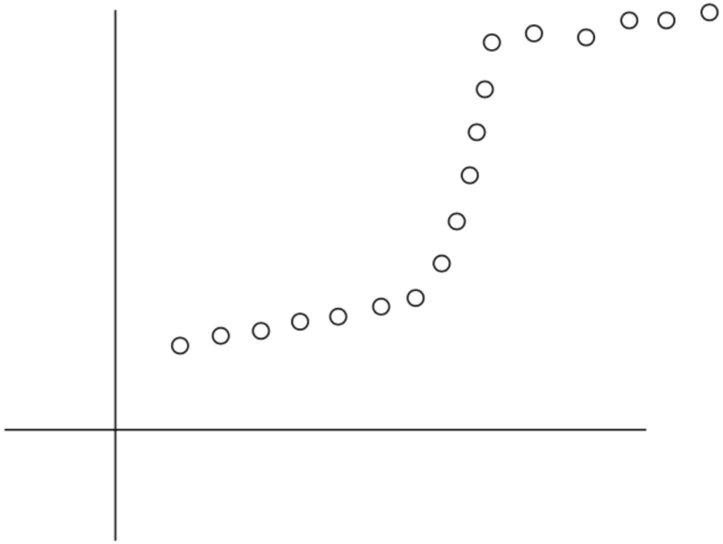
Runtime:

$\leq n$ rec. calls (FindSol)

each call takes $O(n)$ time

TOTAL: $O(n^2)$

SLS: How much space?



1) DP Table: $1 \times (n+1) = O(n)$

Points $\rightarrow O(n)$

$\boxed{\text{Error terms: } O(n^2)}$

Total: $O(n^2)$

M[0]	M[1]	M[2]	…	M[i]	…	M[n]
			…		…	

SLS Wrapup

can solve SLS with a "segment cost" in time $O(n^2)$ space $O(n^2)$

- New idea: multiway case analysis for the final segment