# Analysis of Algorithms, I

## CSOR W4231

Eleni Drinea
*Computer Science Department*

Columbia University

The dynamic programming principle; segmented least squares

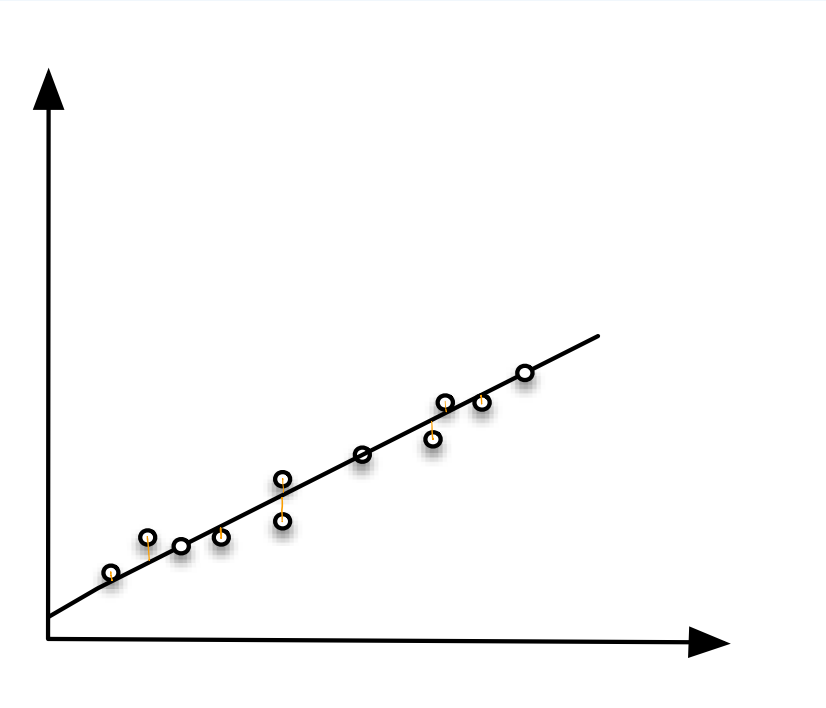
# Outline

# Today

1. Segmented least squares
   - An exponential recursive algorithm
2. A Dynamic Programming (DP) solution
   - A quadratic iterative algorithm
   - Applying the DP principle

# **Linear** least squares fitting

A foundational problem in statistics: find a line of *best fit* through some data points.



# **Linear** least squares fitting

**Input:** a set $P$ of $n$ data points $(x_1, y_1)$, $(x_2, y_2)$, $\ldots$, $(x_n, y_n)$; we assume $x_1 < x_2 < \ldots < x_n$.

**Output:** the line $L$ defined as $y = ax + b$ that minimizes the error

looking for an 'a' and 'b' such that it minimizes the distance to the line.

err (L, P) is a bivariate function in a and b

$$err(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2 \quad (1)$$

minimizing the vertical distances of the data points to the fitted line. Want to find an a & b that minimize the distances.

# **Linear** least squares fitting: solution

Given a set $P$ of data points, we can use calculus to show that the line $L$ given by $y = ax + b$ that minimizes

Given a and b
err(L,P) can be computed in O(n).

$$err(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2 \tag{2}$$

$O(n)$

satisfies

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \tag{3}$$

$O(n)$

Given 'a' you can compute 'b' in O(n) time

$$b = \frac{\sum_i y_i - a \sum_i x_i}{n} \tag{4}$$

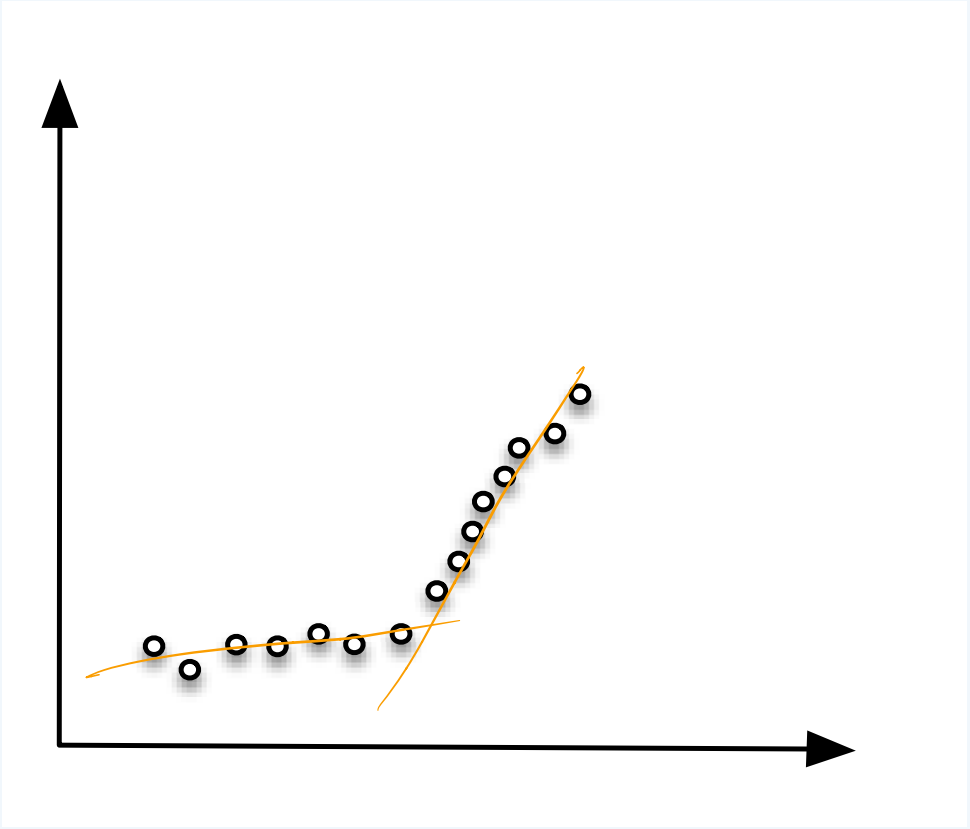
$\longrightarrow O(n)$

*How fast can we compute $a, b$?* $\longrightarrow$ linear time.

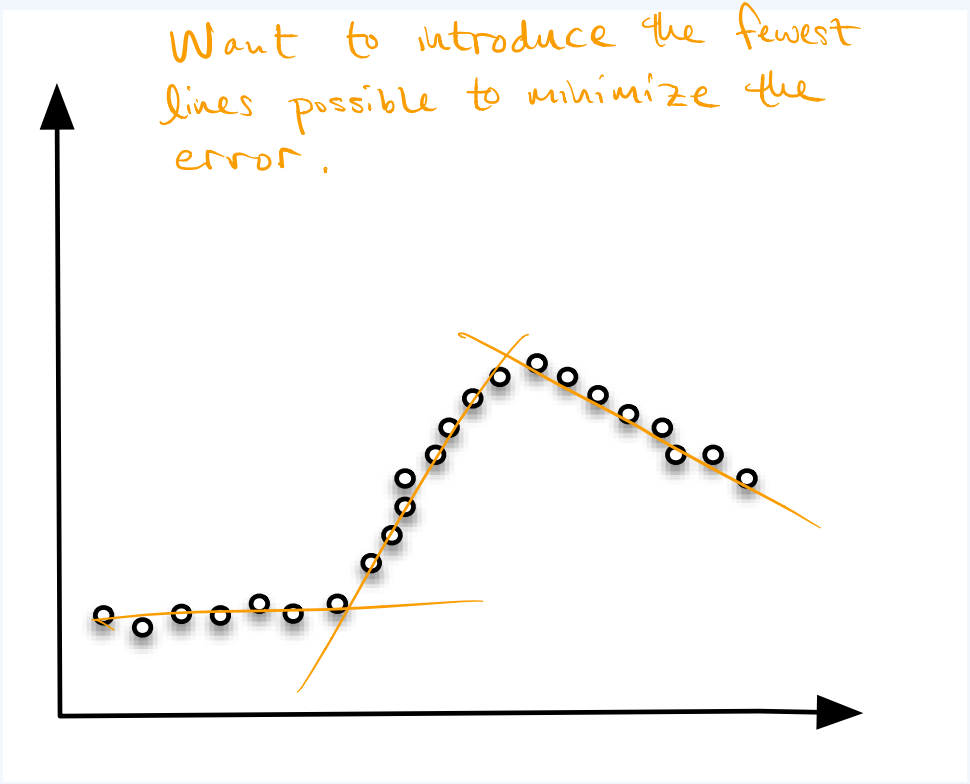
O(n) for 'a' : Use a single for-loop to compute $\Sigma x_i y_i$, $\Sigma x_i$, $\Sigma y_i$, and $\Sigma x_i^2$. Can compute b in O(n) time too.

# *What if the data changes direction?*

- Could fit 2 lines to the data instead of just 1 line to minimize the error w.r.t. the data:



# What if the data changes direction more than once?



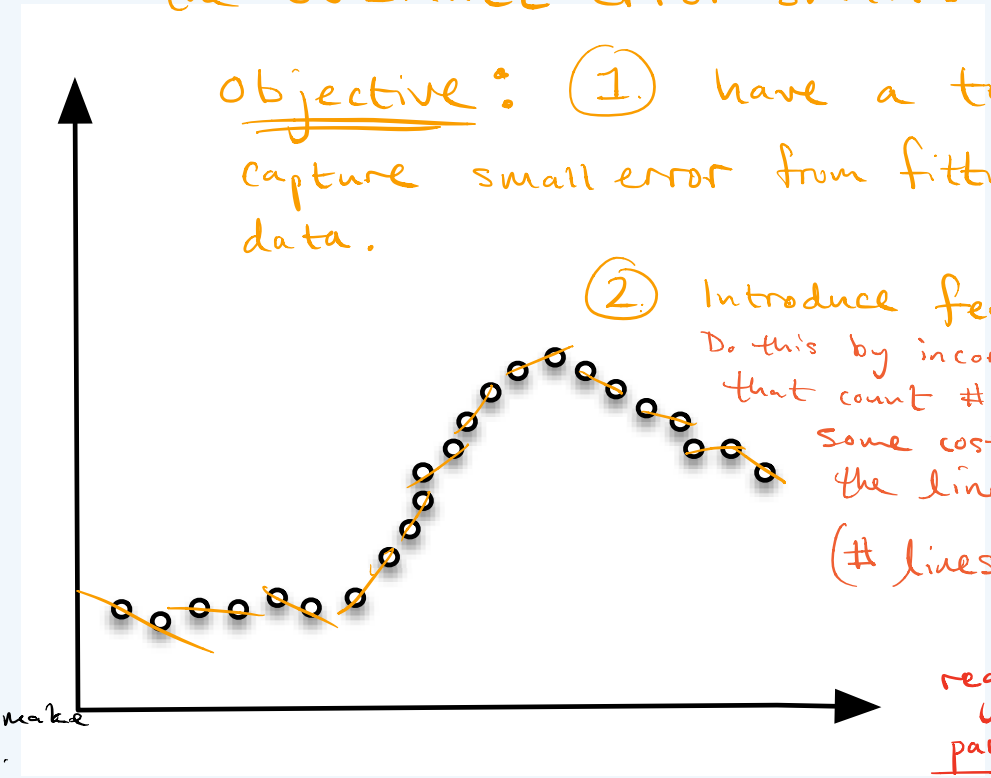

Here, 2 lines will not suffice.
- Will need 3 lines.

However, don't want to look at data and try to guess the # of lines needed to fit the data and hardwire that into your code. We want an algorithm that automatically detects change in direction.

- Want an algorithm that detects change in direction for any input, and therefore introduces new lines as needed to fit the data well.

# What if the data changes direction more than once?

**Real Goal**: Introduce FEW lines, while keeping the OVERALL error small.



**Objective**: (1) have a term to capture small error from fitting lines to data.

(2) Introduce few lines.

Do this by incorporating a term that count # of lines times some cost for introducing the line:

$(\#\ \text{lines}) \cdot C$

$C$ ↑ regularization parameter

Don't want to make objective super hard.

(*) We have 3rd implicit goal which is to come up w/ an optimization problem that we can actually solve efficiently.

Our objective will be a simple sum of the error of fitting lines to the data plus a cost for introducing a new line times the number of lines. :

$$\text{Obj} = \{\text{error for fitting lines to data}\} + \{C \cdot \#\text{ lines}\}$$

where $C$ is the cost for introducing a line. (cost per 1 line).

This captures the trade-off between accuracy and simplicity of the model.

$$\text{Obj} = \underbrace{\{\text{error for fitting lines to data}\}}_{\text{keeping the error small increases the \# of lines}} + \underbrace{\{C \cdot \#\text{ lines}\}}_{\text{keeping overall cost low increases the error.}}$$

$\downarrow$ Error $\Rightarrow \uparrow \{C \cdot \#\text{ of lines}\}$

$\downarrow$ # lines $\Rightarrow \uparrow$ Error

Keeping $C$ small $\Rightarrow$ Care more about accuracy than simplicity. (or smaller error)

Large $C$ $\Rightarrow$ Care more about simplicity of the model because adding lines causes the objective to grow a lot.

//

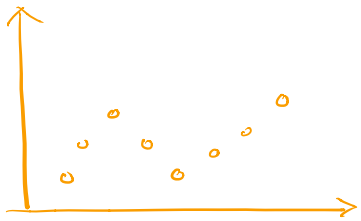
By fine tuning $C$, you can err towards simplicity or increased accuracy for the model.

$C$ is a regularization parameter.

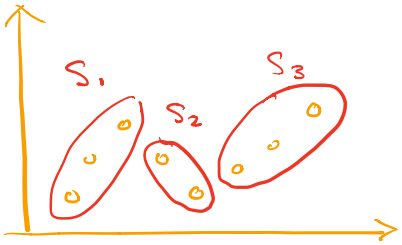
- if you care more about accuracy, keep $C$ small.
- if you care more about simplicity, keep $C$ large.

## What we're doing:

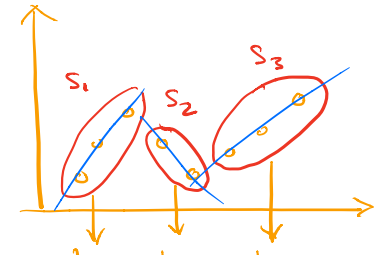
Input data that changes direction:



1. Partition input into segments of consecutive data points




2. Fit lines of best fit to each segment to minimize the total error of fitting lines




$(a_1, b_1)$ $(a_2, b_2)$ $(a_3, b_3)$ ← coefficients of lines $L_1, L_2, L_3$, respectively.

$err(L_1, S_1)$ $err(L_2, S_2)$ $err(L_3, S_3)$

**Segment $S_i$ :** contiguous set of data points $\{p_1, p_2, \ldots, p_j\}$

**Partition or Segmentation of the input P :**

a partitioning of the input into $m$ segments: $S_1, S_2, \ldots, S_m$ (Call this Partition A)

**Cost of Partition A :**

$$\sum_{i=1}^{m} err(L_i, S_i) + m \cdot C$$

$err(L_i, S_i)$: error of line $L_i$ in segment $S_i$

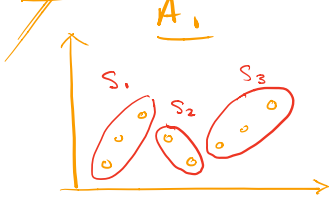
model error

$m \cdot C$: total cost of # of lines used.

($m$ = # of segments or lines)

$m \equiv$ # of segments (and thus the # of lines).

---




Partition $A_1$

$$cost(A_1) = \sum_{i=1}^{3} err(L_i, S_i) + 3C$$

Partition $A_2$


$$\text{Cost}(A_2) = \sum_{i=1}^{2} \text{err}(L_i, S_i) + 2C$$

Specifying a Partition $A$: There are a number of ways. A few ways include:

- By segment of pts: $\{p_1, \dots, p_k\}, \{p_{k+1}, \dots, p_\ell\}, \{p_{\ell+1}, \dots, p_n\}$
- By starting points of each segment:
  $\{p_1, p_6, p_{11}\}$
- By the last points of each segment:
  $\{p_5, p_{10}, p_{20}\} \xrightarrow{\text{equiv.}} \{p_5, p_{10}\}$ ← equivalent since it's clear $p_{20}$ is last.
- # of segments & the end pts: $\{3, p_5, p_{10}, p_{20}\}$

# How to detect change in the data

- Any single line would have large error.
- **Idea 1:** hardcode number of lines to 2 (or some *fixed* $m$).
  - Fails for the dataset on the last slide.
- **Idea 2:** pass an *arbitrary set* of lines through the points and seek the set of lines that minimizes the error.
  - Trivial solution: have a different line pass through each pair of consecutive points in $P$.
- **Idea 3:** fit the points well, using as few lines as possible.
  - Trade-off between complexity and error of the model

# Formalizing the problem

**Input:** data set $P = \{p_1, \ldots, p_n\}$ of points on the plane.

- A segment $S = \{p_i, p_{i+1}, \ldots, p_j\}$ is a contiguous subset of the input.
- Let $\mathcal{A}$ be a partition of $P$ into $m_{\mathcal{A}}$ segments $S_1, S_2, \ldots, S_{m_{\mathcal{A}}}$. For every segment $S_k$, use (2), (3), (4) to compute a line $L_k$ that minimizes $err(L_k, S_k)$.
- Let $C > 0$ be a fixed multiplier. The cost of partition $\mathcal{A}$ is

$$\sum_{S_k \in \mathcal{A}} err(L_k, S_k) + m_{\mathcal{A}} \cdot C$$

# Segmented least squares

This problem is an instance of change detection in data mining and statistics.

**Input:** A set $P$ of $n$ data points $p_i = (x_i, y_i)$ as before.

**Output:** A segmentation $\mathcal{A}^* = \{S_1, S_2, \ldots, S_{m_{\mathcal{A}^*}}\}$ of $P$ whose cost

$$\sum_{S_k \in \mathcal{A}^*} err(L_k, S_k) + m_{\mathcal{A}^*} C$$

is minimum.

| # of segments | | # partitions w/ m segments |
|---|---|---|
| $m = 1$ | 1 2 . . . n<br>o o o . . . o | $1 = \binom{n-1}{0}$ |
| $m = 2$ | 1 \| 2 \| . . . n-1 \| n<br>o \| o \| . . . o \| o<br>↑ \| ↑ \| ↑ | $n-1$<br>$= \binom{n-1}{1}$ |
| | Stopping Points (Recall, pts must be consecutive)<br>* $n-1$ choices to end your first segment | |
| $m = 3$ | 1 2 . . . n-1 n<br>o o o . . . o o | $\binom{n-1}{2}$ |
| | • The last point must be n, and need to find 2 places to end the first two segments, and there are $\binom{n-1}{2}$ choices of this | |
| ⋮ | ⋮ | ⋮ |
| $m = k$ | Need to find k-1 places to end the first k-1 segments. | $\binom{n-1}{k-1}$ |
| ⋮ | ⋮ | ⋮ |
| $m = n$ | 1 \| 2 \| . . . \| n-1 \| n<br>o \| o \| . . . \| o \| o | $1$<br>$= \binom{n-1}{n-1}$ |
| | Each point is its own segment. | |

$$\sum_{k=0}^{n-1} \binom{n-1}{k} = 2^{n-1}$$

Total # of possible partitions

$$= \underline{O}(2^n) = \underline{\Omega}(2^n)$$

---

Recall:

$$\sum_{i=0}^{n} \binom{n}{i} \cdot a^i \cdot b^{n-i} = (a+b)^n$$

For $a = b = 1$:

$$\sum \binom{n}{i} \cdot a^i \cdot b^{n-i} = \sum_{i=0}^{n} \binom{n}{i} = 2^n$$

Brute force algorithm requires exponential time

---

Issue w/ this count:

- No reason to have partitions with $m > \frac{n}{2}$.

  $\frac{n}{2}$ segments means every segment has two data points, and there is only one possible partitioning of $n$ pts into $\frac{n}{2}$ segments of consecutive data points.

$\Rightarrow$ The max # of lines to use is $\frac{n}{2}$ and there is only one possible partition w/ $\frac{n}{2}$ segments. o o | o o | o o | ... | o o

The count however is much larger for $n/2$, but should only be 1.

Need to get more appropriate bound.

Closer to: $\underline{O}(2^{n/2}) = \Omega(2^{n/2})$

(underestimate for the true # of partitions)

---

- Brute force algorithm requires exponential time.

  - Lowerbound of brute force approach is still exponential $\Rightarrow$ Inefficient.

---

NOTE: If you want to show an algorithm is inefficient, you need to show that the lowerbound is exponential (or inefficient)

# A brute force approach

We can find the optimal partition (that is, the one incurring the minimum cost) by exhaustive search.

- Enumerate every possible partition (segmentation) and compute its cost.
- Output the one that incurs the minimum cost.

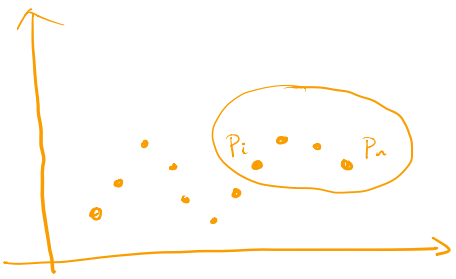
△ $\Omega(2^n)$ partitions

↑ inefficient

**Aim**: Recursive approach/algorithm.

- Extract/Remove the last segment of the optimal segmentation.
- Continue recursively.

→ Let $A^*$ be the optimal segmentation.




- Extract the last segment $S$.
- We know that $p_n \in S$.
- Do not know: where $S$ starts.
- We know that $S$ starts at some point $p_i$, where $p_i$ could be $p_1$ or $p_2$ or $p_3$ or ... or $p_n$ $(1 \leq i \leq n)$

---

**Assume** that $S$ starts at $p_i$:

Let $OPT(n)$ = min cost of segmenting $p_1, \ldots, p_n$

- i.e., the minimum cost of segmenting the entire input.

Can we express this recursively?

$$OPT(n) = C + err(L, \{p_i, \ldots, p_n\}) + OPT(i-1)$$

cost of line in last segment

error of line in last segment

min cost for segmenting $p_1, \dots, p_{i-1}$

cost/contribution of last segment

$$OPT(n) = C_n + err(L, \{p_i, \dots, p_n\}) + OPT(i-1)$$

$OPT(i-1)$: min cost to segment $p_1, \dots, p_{i-1}$

$\min_{1 \le i \le n} \{ \}$

We do not know $p_i$ though — need to find it:

Your $p_i$ should be such that it minimizes the total cost. That is, we minimize over $p_i$ (the starting points):

$$OPT(n) = \min_{1 \le i \le n} \{ C_n + err(L, \{p_i, \dots, p_n\}) + OPT(i-1) \}$$

$C_n + err(L, \{p_i, \dots, p_n\})$: Accounts for pts $p_i, \dots, p_n$

$OPT(i-1)$: still need to segment rest

Minimize over every possible choice for $p_i$

We try every possible $p_i$ and find the one that minimizes the expression.

Boundary condition:

$$OPT(0) = 0$$

This is the recursive algorithm.

To apply recursively to remove the last segment, define:

- $e_{ij} \triangleq err(L, \{p_i, \dots, p_j\})$

PRECOMPUTE ⇕ STORE $e_{ij}$ in $O(n^2)$

$e_{ij} \equiv$ error of fitting a line to the segment that starts at $p_i$

$$OPT(j) = \min_{1 \le i \le j} \{ e_{ij} + C + OPT(i-1) \}$$

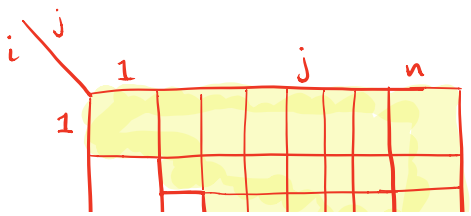
↑ Can access $e_{ij}$ in $O(1)$ time

- We pre-compute the $e_{ij}$ values and store them in a matrix, so that whenever I need to look at them I can do so in constant time.

---

$1 \le i \le j \le n \implies \Theta(n^2)$ $e_{ij}$ terms

- if $i < j$ we have $\binom{n}{2}$ $e_{ij}$ terms.
- if $i \le j$ we have $\binom{n}{2} + n$ $e_{ij}$ terms.




Segments to consider:

- $\{p_1\}, \{p_1, p_2\}, \dots, \{p_1, \dots, p_n\}$
- $\{p_2\}, \dots, \{p_2, \dots, p_n\}$
- ⋮
- ⋮

$E$ =


$e_{ij}$ in time: $O(j-i) = O(n)$

$E$ in time: $O(n^3)$

$\{p_n\}$

Interested in filling in the upper triangular region of this matrix, including the diagonal entries. There are a set $n^2$ entries in total in the matrix $E$.

- Time to fill in $e_{ij} = O(j-i) = O(n)$

$O(n)$ time to compute $a_i$ and $b_i$ for $L_i = a_i x + b_i$. Then $err(L_i, S_i)$ takes $O(n)$ time.

$\Rightarrow$ Total time to fill in $E = O(n^3)$

since there are $n^2$ entries.
and each entry takes $O(n)$ time.

$\rightarrow$ Can be improved to $O(n^2)$

---

PRECOMPUTE & STORE $E$

- time : $O(n^2)$
- space : $O(n^2)$

---

Running Time :

$$OPT(n) = \min_{1 \le i \le n} \{ e_{ij} + C + OPT(i-1) \}$$

Let $T(n) \equiv$ time to compute $OPT(n)$

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \dots$$

$\Omega \equiv \Lambda$

$$\cdots + \cdots \quad T(0) \quad + \quad \underbrace{T(i-1) + \Omega(1)}$$

compute $e_{ij} + C + OPT(i-1)$ for a fixed $i$

And $T(i-1) + O(1) = O(1)$

Need to compute $OPT(i)$ for $1 \le i \le n$:

$$OPT(n) = \min \begin{cases} e_{n,n} + C + OPT(n-1), \\ e_{n,n-1} + C + OPT(n-2), \\ e_{n,n-2} + C + OPT(n-3), \\ \cdots \end{cases}$$

BAD!

$$= T(n-1) + T(n-2) + \cdots + T(0) + \Omega(1)$$

$$\geq T(n-1) + T(n-2)$$ ← Prove that it is super polynomial?

All you need to prove a negative result is to show it has a bad lowerbound.

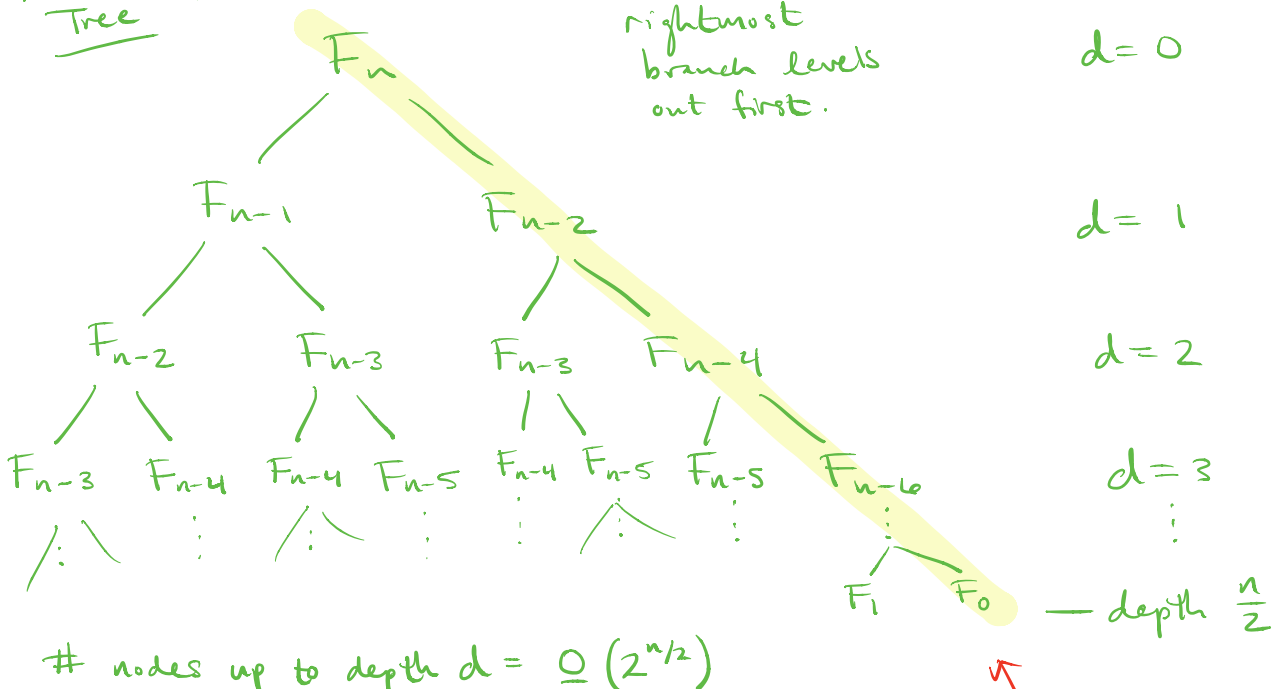
$$T(n) \geq T(n-1) + T(n-2)$$

Look familiar?

Fibbonacci Numbers!

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, \; F_1 = 1$$

Recursion Tree:




rightmost branch levels out first.

$d = 0$

$d = 1$

$d = 2$

$d = 3$

— depth $\frac{n}{2}$

# nodes up to depth $d = \Omega(2^{n/2})$

- At depth $\frac{n}{2}$, the right-most branch bottoms out (first one to bottom out)
- There are $2^{n/2}$ leaves in a complete binary tree.

Other branches not done, but this already requires $\Omega(2^{n/2})$

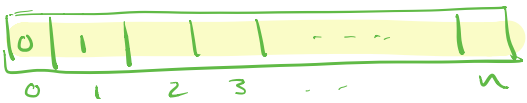
$\Longrightarrow$ If you sum up all the work spent in this tree to get $F_n$, you are summing up at least $2^{n/2} \cdot c$

Therefore, $F_n \geq 2^{n/2}$

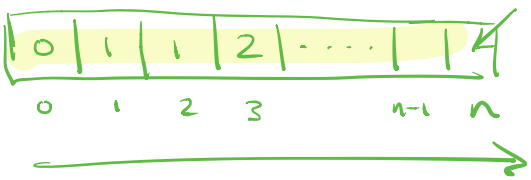
and thus $T(n) = \Omega(2^{n/2})$ ← Super Polynomial time (Inefficient!)

NOTE: Computing the $n^{th}$ Fibbonacci # does NOT require exponential time.

Create an array F with F[0] = 0 and F[1] = 1 : F =




to store the Fibb. #'s, computing $F_k$ from the bottom up. :




Time to compute $F_n$ this way $= O(n)$

(Assuming it requires $O(1)$ time to combine 2 #'s to compute a single value)

- Doing it this way we are only computing one branch of the tree by storing in the array, solutions to subproblems that were already previously computed, so that you can look them up in constant time in the future. We can do this because $F_n = F_{n-1} + F_{n-2}$ has overlapping subproblems.

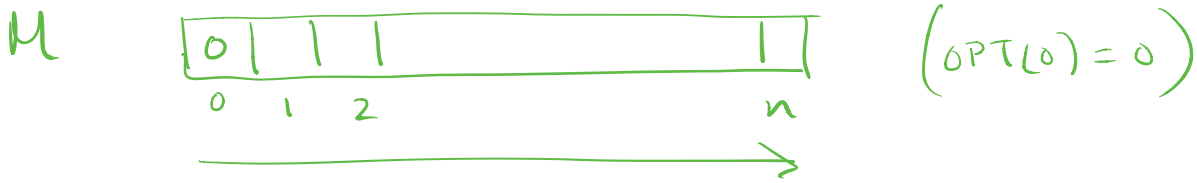
$$OPT(n) = \min_{1 \le i \le n} \{ e_{ij} + C_i + OPT(i-1) \}$$

also has overlapping subproblems, so we can compute this faster using a similar approach described above, using a bottom-up approach. (Generally, this is how we will approach dynamic programming problems :-). We need space to store solutions.

$$OPT(n) = \min_{1 \le i \le n} \{ e_{ij} + C + OPT(i-1) \}$$

Introduce an array $M$ to store solutions.

$M[i] = OPT(i)$ with $n+1$ entries.




$(OPT(0) = 0)$

- Fill it in from the bottom up, using the recurrence.

$$OPT(1) = \cancel{e_{11}}^{0} + C + OPT(0)$$

$$= C$$

$e_{11} = 0$ since fitting line to 1 point.

$$OPT(2) = \min_{1 \le i \le 2} \begin{cases} e_{12} + C + OPT(0), & i=1 \\ e_{22} + C + OPT(1) & i=2 \end{cases}$$

$$= \min_{1 \le i \le 2} \{ C, 2C \} = C$$

$$OPT(3) = \min_{1 \le i \le 3} \begin{cases} e_{13} + C + OPT(0), \\ e_{23} + C + OPT(1), \\ e_{33} + C + OPT(2) \end{cases}$$

$$= \min_{1 \le i \le 3} \{ e_{i3} + C, 2C, 3C \}$$

$$= X$$

$M =$

| 0 | C | C | X | ... | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | | n |

---

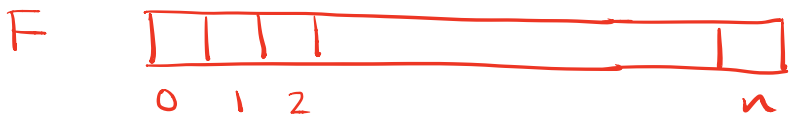
How can we prove correctness ?

Induction (Do on our own as an exercise)

- EASY TO COMPUTE RECURRENCE

$$F_n = F_{n-1} + F_{n-2}$$

- STORE & RE-USE solutions to subproblems




- ITERATIVE BOTTOM-UP COMPUTATIONS

$F[0] = 0$

$F[1] = 1$

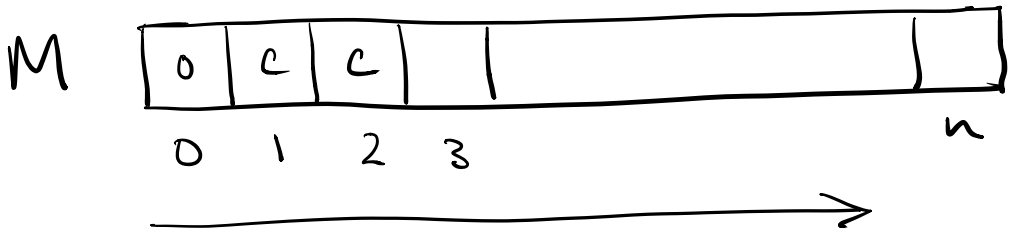
$F[i] = F[i-1] + F[i-2]$ for $i \geq 2$

- We want : $F_n$
- Time per subproblem : $O(1)$
- # subproblems : $n$
- Total time : $O(n)$
- Space : $O(n)$

$$OPT(j) = \min_{1 \le i \le j} \left\{ e_{i,j} + C + OPT(i-1) \right\}$$

$M[j] = OPT(j)$

Computed in a bottom-up fashion.



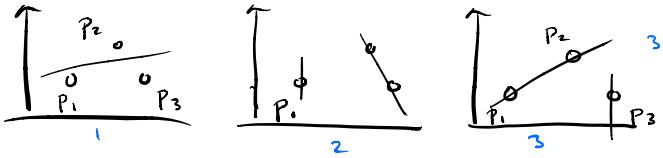

$M[0] = OPT(0) = 0$

$M[1] = OPT(1) = C$

$M[2] = OPT(2) = \min \left\{ \begin{array}{l} e_{12} + C + OPT(0), \\ e_{22} + C + OPT(1) \end{array} \right\}$

$= \min \{ C, 2C \} = C$

$M[3] = OPT(3) = \min \left\{ \begin{array}{l} e_{13} + C + OPT(0) = \\ e_{23} + C + OPT(1) = 2C \\ e_{33} + C + OPT(2) = 2C \end{array} \right.$




$M[4] = OPT(4) =$

$$OPT(j) = \min_{1 \le i \le j} \left\{ e_{i,j} + C + OPT(i-1) \right\}$$

Time for $OPT(j)$ : $O(j)$
$= O(n)$

- We want : $M[n] = OPT(n)$
- Time per subproblem : $O(n)$ ← Because minimizing over $j$ entries.
- # subproblems : $n$
- Total time : $O(n^2)$
- Total space : $O(n)$ + $O(n^2)$

$O(n)$: for dynamic portion of storage ($M$)

$O(n^2)$: for $e_{i,j}$ matrix

# A crucial observation regarding the last data point

Consider the last point $p_n$ in the data set.

- $p_n$ belongs to a single segment in the optimal partition.
- That segment starts at an earlier point $p_i$, for some $1 \leq i \leq n$.

This suggests a recursive solution: **if** we knew where the last segment starts, then we could remove it and recursively solve the problem on the remaining points $\{p_1, \ldots, p_{i-1}\}$.

# A recursive approach

- Let $OPT(j)$ = minimum cost of a partition of the points $p_1, \ldots, p_j$.
- Then, if the last segment of the optimal partition is $\{p_i, \ldots, p_n\}$, the cost of the optimal solution is

$$OPT(n) = err(L, \{p_i, \ldots, p_n\}) + C + OPT(i-1).$$

- But we don't know where the last segment starts! *How do we find the point $p_i$?*
- Set

$$OPT(n) = \min_{1 \le i \le n} \Big\{ err(L, \{p_i, \ldots, p_n\}) + C + OPT(i-1) \Big\}.$$

# A recurrence for the optimal solution

**Notation:** let $e_{i,j} = err(L, \{p_i, \ldots, p_j\})$, for $1 \leq i \leq j \leq n$. Then

$$OPT(n) = \min_{1 \leq i \leq n} \Big\{ e_{i,n} + C + OPT(i-1) \Big\}.$$

If we apply the above expression recursively to remove the last segment, we obtain the recurrence

$$OPT(j) = \min_{1 \leq i \leq j} \Big\{ e_{i,j} + C + OPT(i-1) \Big\} \qquad (5)$$

Time to compute $OPT(j) \longrightarrow \Theta(j) = O(n)$ (and there are $n+1$ subproblems)

**Remark 1.**

1. *We can precompute and store all* $e_{i,j}$ *using equations* (2), (3), (4) *in* $O(n^3)$ *time. Can be improved to* $O(n^2)$.
2. *The natural recursive algorithm arising from recurrence* (5) *is* **not** *efficient (think about its recursion tree!).*

$\Rightarrow$ Time to compute $M$ (entire algorithm): $n \cdot O(n) = O(n^2)$

# Exponential-time recursion

$T(n) \geq F_n \geq 2^{n/2}$

**Notation:** $T(n) =$ time to compute $OPT(n)$, that is, the cost of the optimal partition for $n$ points.

Then

$$T(n) \geq T(n-1) + T(n-2).$$

- Can show that $T(n) \geq F_n$, the $n$-th Fibonacci number (by strong induction on $n$).
- From optional problem 6a in Homework 1, $F_n = \Omega(2^{n/2})$.
- Hence $T(n) = \Omega(2^{n/2})$.

$\Rightarrow$ The recursive algorithm requires $\Omega(2^{n/2})$ time.

# Today

1. Segmented least squares
   - An exponential recursive algorithm

2. A Dynamic Programming (DP) solution
   - A quadratic iterative algorithm
   - Applying the DP principle

# Are we really that far from an efficient solution?

Recall Fibonacci problem from HW1: exponential recursive algorithm, polynomial iterative solution

*How?*

1. Overlapping subproblems: spectacular redundancy in computations of recursion tree
2. Easy-to-compute recurrence for combining the smaller subproblems: $F_n = F_{n-1} + F_{n-2}$
3. Iterative, bottom-up computations: we computed and stored the subproblems from smallest $(F_0, F_1)$ to largest $(F_n)$, iteratively.
4. Small number of subproblems: only solved $n - 1$ subproblems.

# Elements of DP in segmented least squares

Our problem exhibits similar properties.

1. Overlapping subproblems
2. Easy-to-compute recurrence for combining optimal solutions to smaller subproblems into the optimal solution of a larger subproblem (once smaller subproblems have been solved)
3. Iterative, bottom-up computations: compute the subproblems from smallest (0 points) to largest ($n$ points), iteratively.
4. Small number of subproblems: we only need to solve $n$ subproblems.

# A dynamic programming approach

$$OPT(j) = \min_{1 \le i \le j} \left\{ e_{i,j} + C + OPT(i-1) \right\}$$

- The optimal solution to the subproblem on $p_1, \ldots, p_j$ contains optimal solutions to smaller subproblems.
- Recurrence 5 provides an **ordering** of the subproblems from smaller to larger, with the subproblem of size 0 being the smallest and the subproblem of size $n$ the largest.

$\Rightarrow$ There are $n+1$ subproblems in total. Solving the $j$-th subproblem requires $\Theta(j) = O(n)$ time.

$\Rightarrow$ The overall running time is $O(n^2)$.

- Boundary conditions: $OPT(0) = 0$.
- Segment $p_k, \ldots, p_j$ appears in the optimal solution only if the minimum in the expression above is achieved for $i = k$.

# An iterative algorithm for segmented least squares

This algorithm computes the optimal cost.

Let $M$ be an array of $n$ entries such that

$$M[i] = \text{cost of optimal partition of the first } i \text{ data points}$$

```
SegmentedLS(n, P)
  M[0] = 0
  for all pairs i ≤ j do
      Compute e_{i,j} for segment p_i, ..., p_j using (2), (3), (4)
  end for
  for j = 1 to n do
      M[j] = min_{1≤i≤j} {e_{i,j} + C + M[i − 1]}
  end for
  Return M[n]
```

Precompute and store the $e_{ij}$'s

The heart of the Dynammic Programming algorithm.

**Running time:** time required to fill in dynamic programming array $M$ is $O(n^3) + O(n^2)$. Can be brought down to $O(n^2)$.

We have $n+1$ subproblems, each requires $O(n)$ time.

# Reconstructing an optimal segmentation

$O(n^2)$ is tight

Since this is an optimization problem, we also want to output an optimal solution (optimal segmentation here).

We can reconstruct the optimal partition recursively, using array $M$ and error matrix $e$.

`OPTSegmentation`$(j)$ ← (initially call it with argument $n$)

**if** $(j == 0)$ **then** return

**else**

Extracts each segment.

- Find $1 \leq i \leq j$ such that $M[j] = e_{i,j} + C + M[i-1]$
- `OPTSegmentation`$(i-1)$
- Output segment $\{p_i, \ldots, p_j\}$

Finds the point where the last segmentation in the optimal segment up to point $j$ starts.

**end if**

- Initial call: `OPTSegmentation`$(n)$
- *Running time?*

up to $n/2$ segments

- At most $O(n)$ recursive calls
- Time spent per recursive call: $O(j) = O(n)$

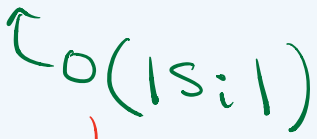
$O(n^2)$ time

# Reconstructing an optimal segmentation

We can reconstruct the optimal partition recursively, using array $M$ and error matrix $e$.

`OPTSegmentation`$(j)$

**if** $(j == 0)$ **then** return
**else**
  Find $1 \leq i \leq j$ such that $M[j] = e_{i,j} + C + M[i-1]$
  `OPTSegmentation`$(i-1)$
  Output segment $\{p_i, \ldots, p_j\}$
**end if**


- Initial call: `OPTSegmentation`$(n)$
- *Running time?*

- We are usually stingier with space than we are with time, unless it helps improve the running time of the algorithm.

- To give the optimal solution, we typically use a recursive algorithm.

---


- might spend a long time to find the i where the last segment starts.

  - Start from the opposite direction. Instead of starting 'from 1 to i', start from point n and go down looking for point i, then you are guaranteed to spend time proportional to the length of the segment to find its starting point. Do this for every segment, so total time spent is proportional to the sum of the length of all the segments, which is n.

This modification guarantees linear time.

# Obtaining efficient algorithms using DP

1. Optimal substructure: the optimal solution to the problem contains optimal solutions to the subproblems.
2. A recurrence for the overall optimal solution in terms of optimal solutions to appropriate subproblems. The recurrence should provide a natural ordering of the subproblems from smaller to larger and require polynomial work for combining solutions to the subproblems.
3. Iterative, bottom-up computation of subproblems, from smaller to larger.
4. Small number of subproblems (polynomial in $n$).

Time for DP algorithms: $O(\#\text{subproblems} \cdot \text{time per subproblem})$

# For DP you need:

1. Recurrence & subproblems

   (- need to define subproblems clearly
   Ex: In this problem, OPT(n) ≡ min cost to segment end pts.)

2. Boundary conditions.

   Ex: In this problem OPT(0) = 0

3. Order of subproblems from smaller to larger.

4. Iterative computation.

5. Space: O(# subproblems)

6. Time: O(# subproblems · time per subproblem)

★ Solution to the original problem appear somewhere in the DP table, or somehow be derived from subproblems.

HW # 3 deals w/ the latter form: Need to derive the solution to the overall problem by looking at some entries in your DP table – you will not find a single entry to give the solution to the

# Dynamic programming vs Divide & Conquer

- They both combine solutions to subproblems to generate the overall solution.
- However, divide and conquer starts with a large problem and divides it into small pieces.
- While dynamic programming works from the bottom up, solving the smallest subproblems first and building optimal solutions to steadily larger problems.