Week 9 - Monday

COMP 4500

Last time

- What did we talk about last time?
- Weighted interval scheduling

Questions?

Assignment 5

Logical warmup

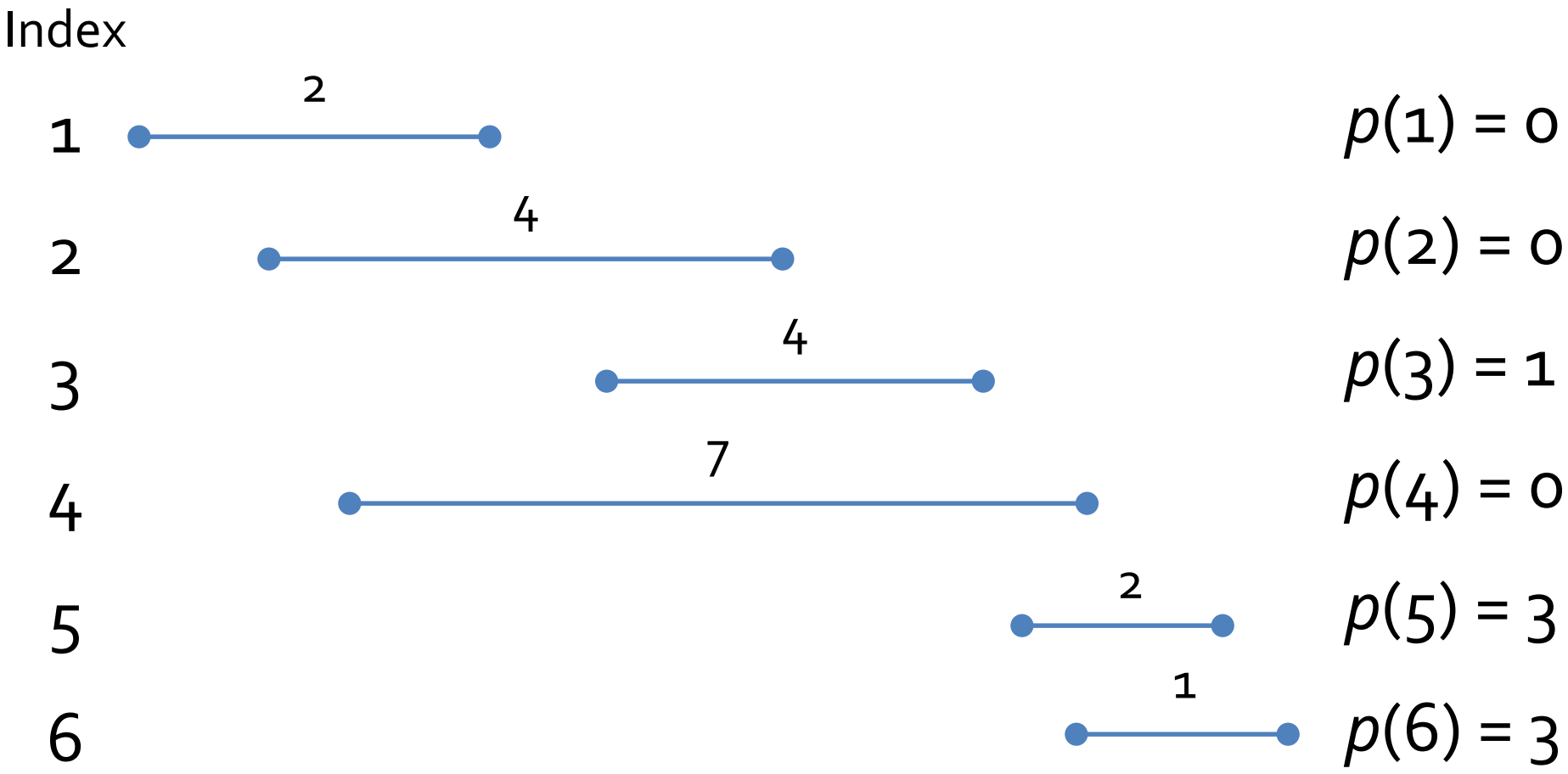
- There are five pirates who wish to divide 100 gold coins
- These pirates are ranked:
  - Captain
  - Lieutenant
  - Master
  - Midshipman
  - Seaman
- In order of rank, each pirate gets the opportunity to propose a plan for dividing up the gold
- If at least half of the pirates (including the proposer) agree on the proposition, it is carried out
- Otherwise, the pirate is killed and the next highest ranking pirate makes a proposal
- Pirates are completely rational, who value, in this order:
  - Staying alive
  - Maximizing gold coins received
  - Seeing other pirates die
- If you were the captain, what would you propose?
- Hint: Work backwards!

Back to Weighted Interval Scheduling

Weighted interval scheduling

- The **weighted interval scheduling** problem extends interval scheduling by attaching a weight (usually a real number) to each request
- Now the goal is not to maximize the **number** of requests served but the total **weight**
- Our greedy approach is worthless, since some high value requests might be tossed out
- We could try all possible subsets of requests, but there are **exponential** of those
- **Dynamic programming** will allow us to save parts of optimal answers and combine them efficiently

$p(j)$ examples



More algorithm design

- Consider an optimal solution ***O***
  - It either contains the last request ***n*** or it doesn't
- If ***O*** contains ***n***, it does not contain any requests between ***p***(***n***) and ***n***
- Furthermore, if ***O*** contains ***n***, it has an optimal solution for the problem for just requests 1, 2, …, ***p***(***n***)
  - Since those requests don't overlap with ***n***, they have to be the best or they wouldn't be optimal
- If ***O*** does not contain ***n***, then ***O*** is simply the optimal solution of requests 1, 2,…, ***n*** - 1

Subproblems found!

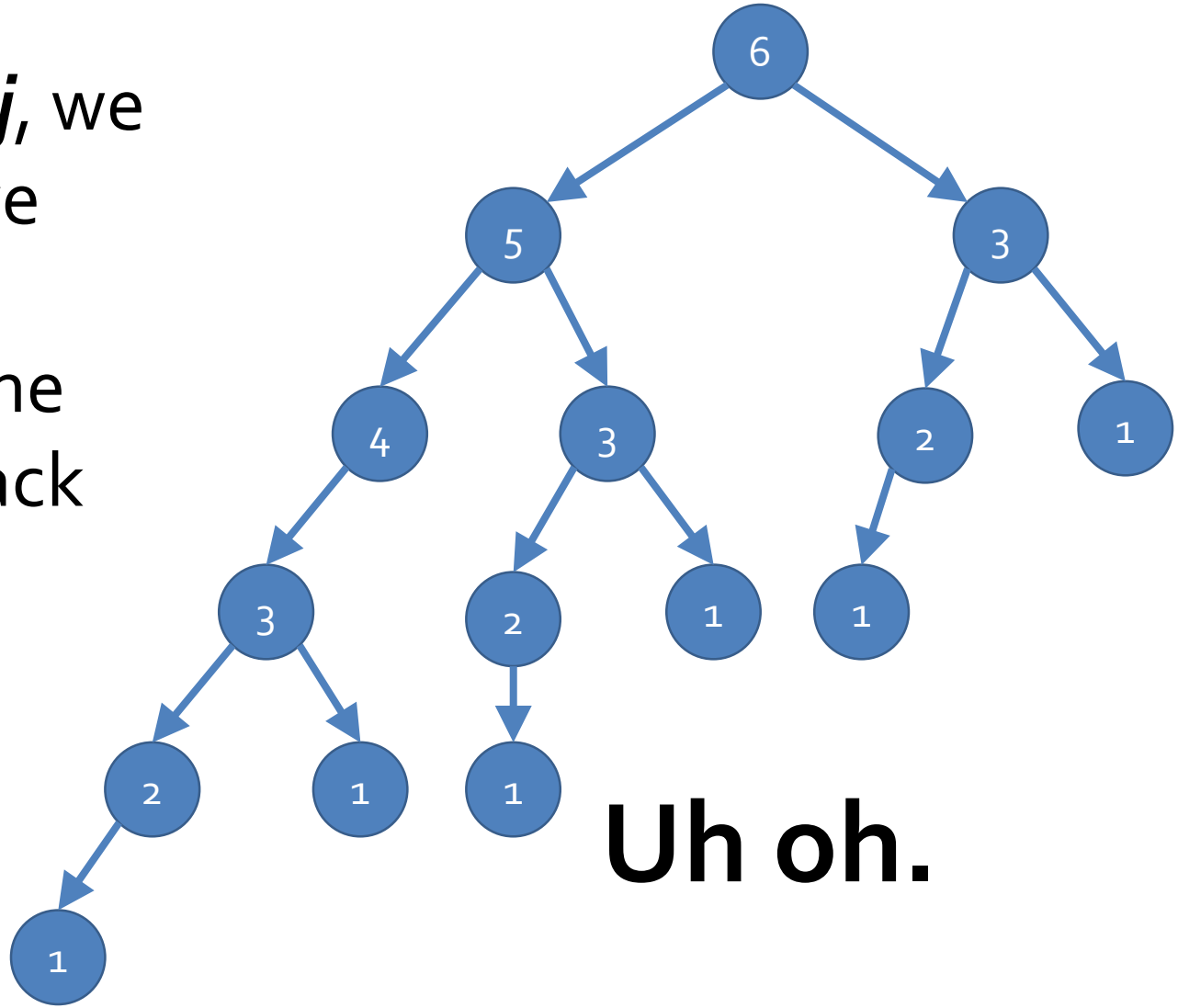
- It might not be obvious, but the last slide laid out a way to break a problem into smaller subproblems
- Let OPT(j) be the value of the optimal solution to the subproblem of requests 1, 2,…, j
- $\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1))$
- Another way to look at this is that we will include j in our optimal solution for requests 1, 2,…,j iff $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$

We've already got an algorithm!

- Compute-Opt(j)
  - If $j = 0$ then
    - Return 0
  - Else
    - Return max(v_j + Compute-Opt($p(j)$), Compute-Opt($j - 1$))

How long does Compute-Opt take?

- Well, for every request ***j***, we have to do two recursive calls
- Look at the tree from the requests a few slides back



Needless recomputation

- The issue here is that we are needlessly recomputing optimal values for smaller subproblems
- You might recall that we had a similar problem in COMP 2100 with the naïve implementation of a recursive Fibonacci function
- In the worst case, the algorithm has an exponential running time
- Just **how** exponential depends on the structure of the problem

Memoization

- The solution is something called **memoization**, which means storing the value for an optimal solution whenever you compute it
- Then, if you need it again, you just look it up (from the *memo* you left yourself)
- To make this work, we need an array ***M*** of length ***n*** that stores the optimal value found for each request
  - Initially, it's all -1 or null or another value that indicates empty

Updated algorithm

- M-Compute-Opt(j)
  - If $j = 0$ then
    - Return 0
  - Else if $M[j]$ is not empty then
    - Return $M[j]$
  - Else
    - $M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$
    - Return $M[j]$

Running time of memoized algorithm

- Constant non-recursive work is done inside of each call
- The recursion will be constant if $M[j]$ already has a value
- There will only be n cases when $M[j]$ doesn't have a value
- The running time is $O(n)$
- Note that sorting the requests in the first place takes $O(n \log n)$

Going beyond the value

- We have only found the *value* of an optimal solution, not the *actual intervals* included
- As with many dynamic programming solutions, the value is the hard part
- For each optimal value, we could keep the solution at that point, but it would be linear in length
- Storing that solution would require $O(n^2)$ space and make the algorithm $O(n^2)$ in order to keep it updated

Reconstructing the solution

- Recall that j is in our optimal solution if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$
- On that basis, we can backtrack through the M array and output request j only if the inequality holds

Algorithm for solution

- Find-Solution(j, M)
  - If $j = 0$ then
    - Output nothing
  - Else if $v_j + M[p(j)] \geq M[j-1]$ then
    - Output j together with the result of Find-Solution($p(j)$)
  - Else
    - Output the result of Find-Solution($j-1$)
- Algorithm is $O(n)$

Three-sentence Summary of Principles of Dynamic Programming and Segmented Least Squares

Student Lecture

Principles of Dynamic Programming

Why is this dynamic programming?

- The key element that separates dynamic programming from divide-and-conquer is that you have to *keep* the answers to subproblems around
- It's not simply a one-and-done situation
- Based on which intervals overlap with which other intervals, it's hard to predict when you'll need an earlier $M[j]$ value
- Thus, dynamic programming can often give us polynomial algorithms *but* with linear (and sometimes even larger) space requirements

Iterative vs. recursive

- The array ***M*** is thus the key to this (and other similar) dynamic programming algorithms
- The same problem could have been tackled with a non-recursive approach where you compute each ***M***[***j***] in order
- Every dynamic programming problem can use either:
  - Memoized recursion
  - Building up solutions iteratively
- The two solutions are equivalent, but the book will prefer iterative solutions, since they are usually faster in practice and easier to analyze

Iterative solution to weighted interval scheduling

- Iterative-Compute-Opt
  - $M[0] = 0$
  - For $j = 1$ up to n
    - $M[j] = \max(v_j + M[p(j)], M[j-1])$

- Algorithm is (even more obviously) $O(n)$

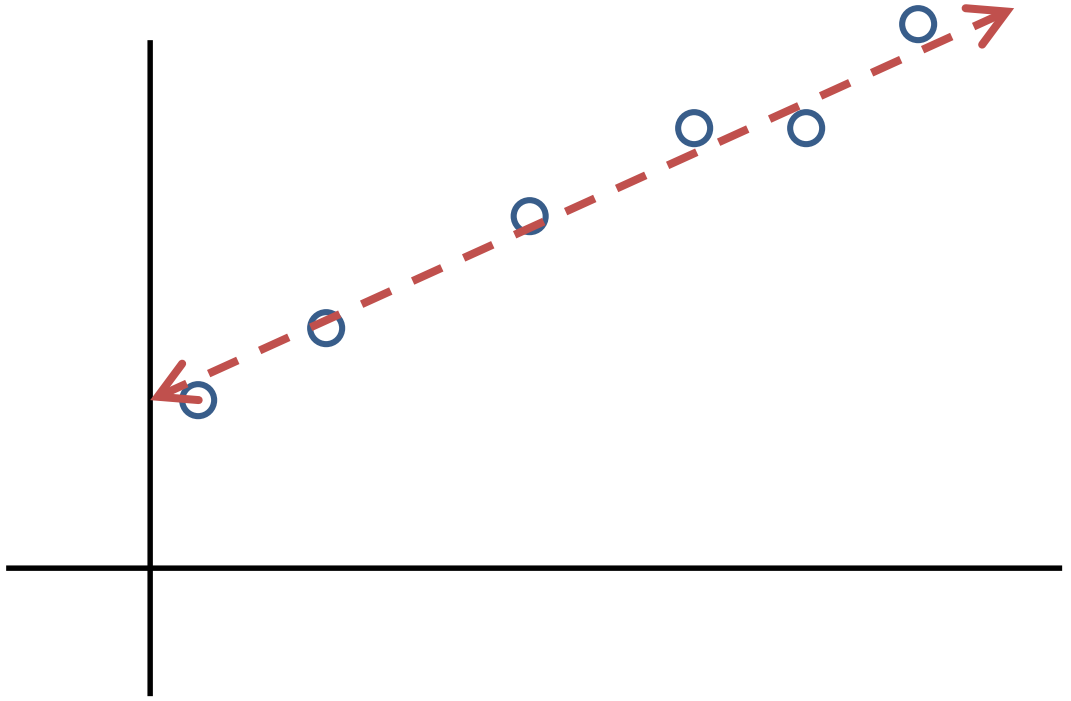
Informal guidelines

- Weighted interval scheduling follows a set of informal guidelines that are essentially universal in dynamic programming solutions:
  1. There are only a polynomial number of subproblems
  2. The solution to the original problem can easily be computed from (or is one of) the solutions to the subproblems
  3. There is a natural ordering of subproblems from "smallest" to "largest"
  4. There is an easy-to-compute recurrence that lets us compute the solution to a subproblem from the solutions of smaller subproblems

Segmented Least Squares

A line of best fit

- Given some data, it is easy to construct a line of best fit



What does such a line look like?

- Consider a set P of n points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
- We want to define a line L as $y = ax + b$
- Such that its error is minimized

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

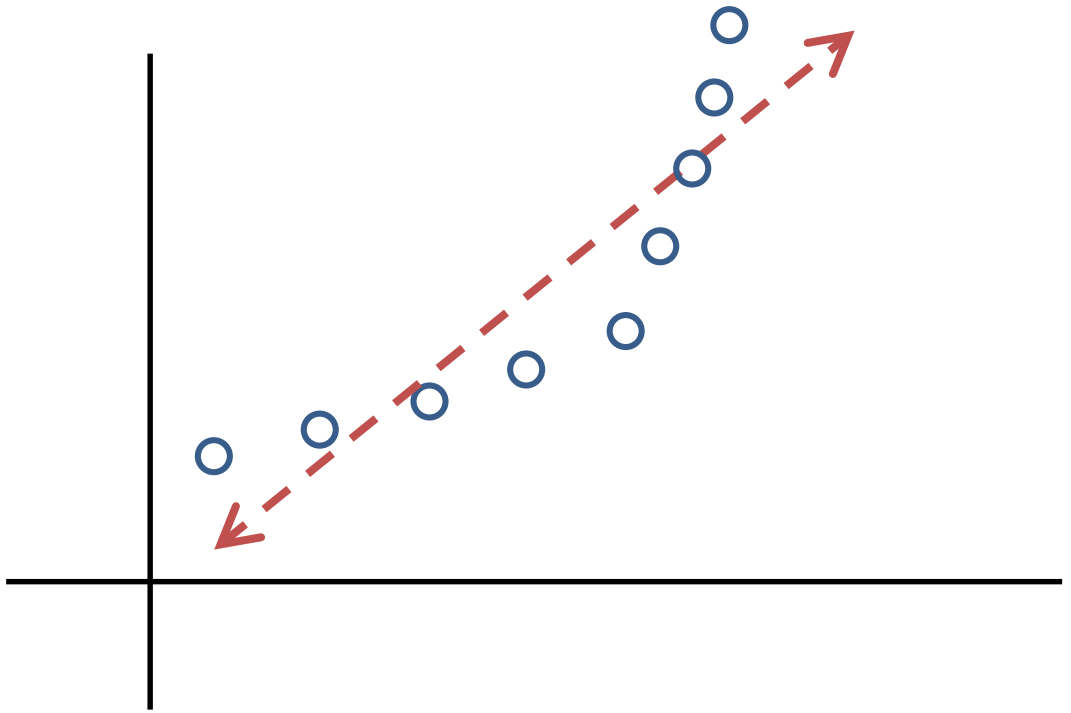
Finding such a line

- Using calculus, it's possible to derive an equation to determine ***a*** and ***b***, the slope and ***y***-intercept of such a line:

$$a = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

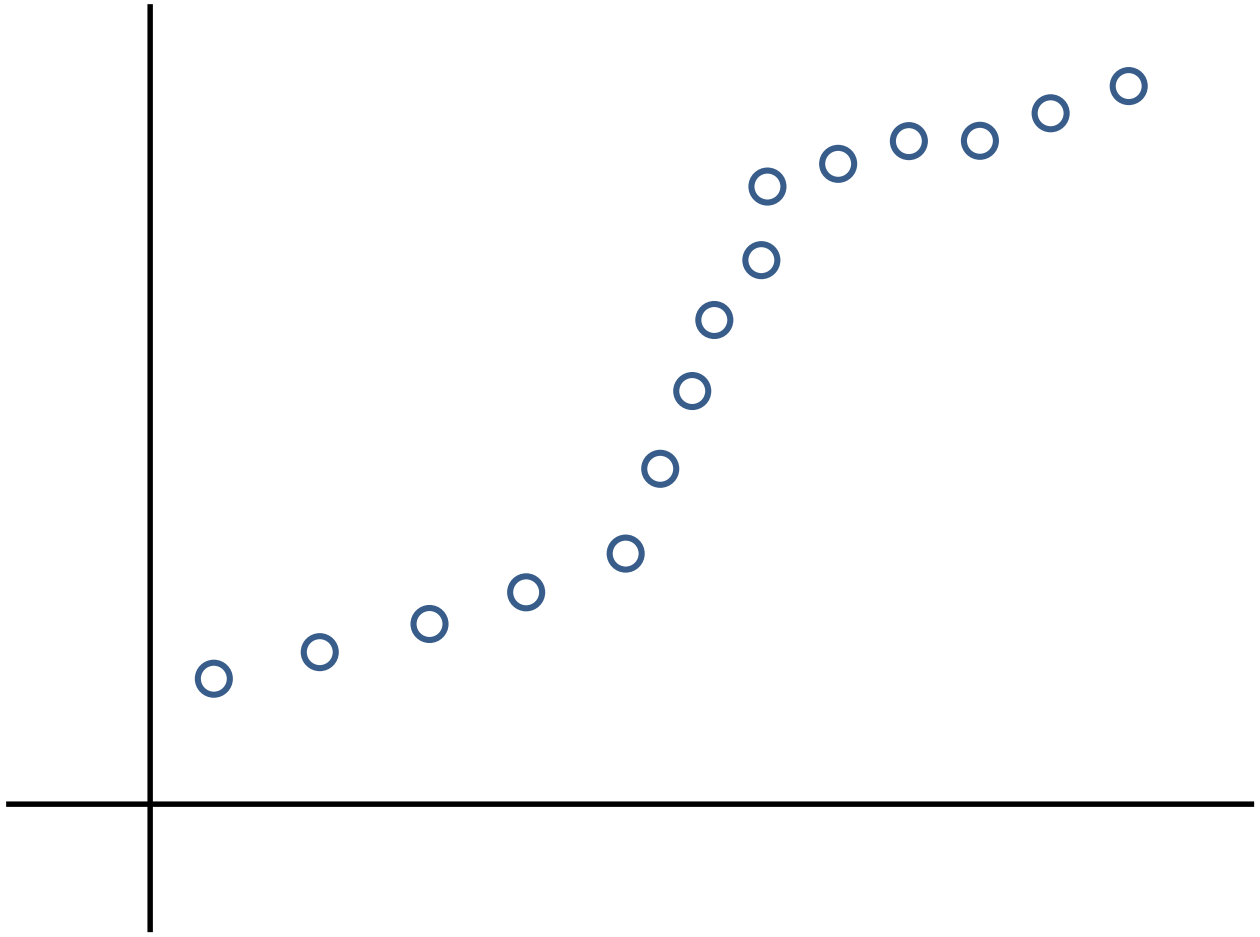
$$b = \frac{\sum_{i=1}^{n} y_i - a\sum_{i=1}^{n} x_i}{n}$$

But what if the data really falls on two lines?



- A single line would have terrible error
- But how do we know that there are two lines?

Or three lines?



We can't allow *any* number of lines

- If we only care about error, $n - 1$ lines will always be the best
  - We could just put a line between each adjacent pair of points and have **no error**
- But that's obviously stupid
- Somehow, we need an algorithm that gives us the intuitively correct number of lines
- We need a **penalty** for adding more lines

Formulating the problem

- We get a set P of n points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
- For simplicity, $x_1 < x_2 < \ldots < x_n$
- Let p_i be (x_i, y_i)
- We want to partition P into **segments**
- Each segment is a contiguous set of x-coordinates:
  - $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for indexes $i \leq j$

Minimizing the penalty

- Using the formula from earlier, we compute lines minimizing the error for each segment
- The penalty of the whole partition is the sum of:
  - The number of segments multiplied by some constant $C > 0$
  - The error value of the optimal lines through each segment
- C is a constant provided to the algorithm
  - The higher the C, the greater then penalty for having more segments

Algorithm design

- How many ways are there to partition a list of n items?
  - 2^{n-1}
  - We can't consider all possible partitions
- Somehow, we need to divide the problem into smaller subproblems
- The last point p_n is in the last segment, whatever it is
- That segment starts at some point p_i
- If we knew the last segment, we could recursively solve the problem on points $p_1, p_2, \ldots, p_{i-1}$

More algorithm design

- Let OPT(i) be the optimum solution for points $p_1, p_2, \ldots, p_i$
- Let $e_{i,j}$ be the minimum error of any line with respect to $p_i, p_{i+1}, \ldots, p_j$
- It must be the case that, if the last segment of the optimal partition is $p_i, p_{i+1}, \ldots p_n$, then the value of the optimal solution is:
  - $\text{OPT}(n) = e_{i,n} + C + \text{OPT}(i-1)$

Final recurrence

- We can generalize that observation for any subproblem going up to point p_j:

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

- Also, the segment $p_i, p_{i+1}, \ldots, p_j$ is used in an optimum solution for the subproblem if and only if the minimum is obtained with index ***i***
- Note that we assume OPT(0) = 0

Algorithm

- Segmented-Least-Squares(n)
  - Create array M with indexes 0 through n
  - Set $M[0] = 0$
  - For all pairs $i \leq j$
    - Compute the least squares error $e_{i,j}$ for segment $p_i, p_{i+1}, \ldots p_j$
  - For $j = 1$ up to n
    - For $i = 1$ up to j
      - Set $M[j] = \min(M[j], e_{i,j} + C + M[i-1])$
  - Return $M[n]$

Reconstructing the segments

- As before, we only find the total penalty value with the previous algorithm, not the actual segments themselves
- We use the observation that the segment $p_i, p_{i+1}, \ldots, p_j$ is used in an optimum solution for a subproblem if and only if the minimum was obtained using index i
- Note that this minimal i could be recorded during the initial algorithm

Algorithm for segments

- Find-Segments(***j***)
  - If ***j*** = 0 then
    - Output nothing
  - Else
    - Find an ***i*** that minimizes $e_{i,j} + C + M[i-1]$
    - Output the result of Find-Segments(***i*** – 1) and the segment $\{p_i, p_{i+1}, \ldots, pj\}$

Running time

- First, we have to compute the values of the least-squares errors $e_{i,j}$
  - There are $O(n^2)$ pairs (i,j)
  - It takes $O(n)$ time to compute each least-squares error for each pair
  - The total is $O(n^3)$, but there are tricks that can get it down to $O(n^2)$
- The algorithm has nested For loops, for j from 1 up to n and for i from 1 up to j, a total of $O(n^2)$
- Let k be the number of segments
  - Reconstructing the segments takes $O(kn)$ time, unless the minimal i values are recorded, in which case it takes $O(k + n)$ time, which is $O(n)$
- Thus, the total is $O(n^3)$ unless using tricks to get the error computation down to $O(n^2)$

Quiz

Upcoming

Next time…

- Subset sums and knapsacks

Reminders

- Read section 6.4
- Start Assignment 5