Bonus Topic - Dynamic Programming

Idea: Find optimal solutions by solving subproblems, then building up bit-by-bit.
kind of like the branching in B+B

Weighted Interval Scheduling

n requests

Brute Force: $O(2^n)$
Greedy: $O(n)$, but not optimal
Backtracking/B+B: $O(2^n)$

Dynamic Programming: $O(n)$ and optimal
Linear time! Crazy!

Suppose we have requests
$$R = \{r_1, r_2, r_3, \dots, r_n\}$$
Each request r_i has a value v_i, start time s_i and finish time f_i. Assume we have sorted by finish time: $f_1 \le f_2 \le \dots \le f_n$.

Given any set S of requests, define $O(S)$ to be the score of the optimal solution using intervals in S.

Define $R_k = \{r_1, r_2, r_3, \dots, r_k\}$ $(k \le n)$

Question: If we know $O(R_1), O(R_2), \ldots, O(R_{\ell-1})$, then can we use this to compute $O(R_\ell)$?

$O(\{r_1, r_2\})$

If the answer is "yes", then we're in good shape!

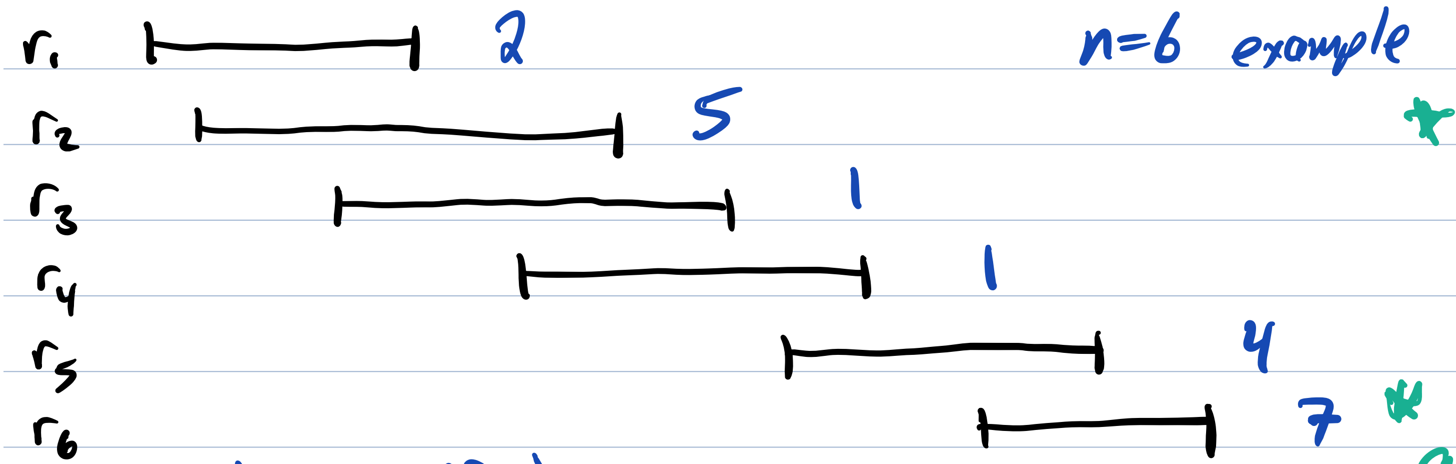
→ $O(R_1)$ is easy to compute. $O(\{r_1\}) = v_1$

→ Use $O(R_1)$ to compute $O(R_2)$.

→ Use $O(R_1)$ and $O(R_2)$ to compute $O(R_3)$.

⋮

→ Use $O(R_1), \ldots, O(R_{n-1})$ to compute $O(R_n)$.



Want $O(R_6)$

If we know $O(R_1) = 2$, $O(R_2) = 5$, $O(R_3) = 5$, $O(R_4) = 5$, $O(R_5) = 9$, can we easily get $O(R_6)$?

Fact: r_6 is either [in an optimal solution] or [not in any optimal solution].

If not: $O(R_6) = O(R_5)$

If so: $O(R_6) = \underset{7}{r_6} + O(R_4)$

Formula: $O(R_6) = \max(O(R_5), 7+O(R_4))$

$O(R_6) = \max(9, 7+5) = 12$

This is a recursive formula. $= \max(9, 7+5) = 12$!

$$O(R_6) = \max(O(R_5), 7+O(R_4))$$

$$\overset{=9}{O(R_5)} = \max(O(R_4), 4+O(R_3)) = \max(5,9) = 9 \qquad \overset{=5}{O(R_4)} = \max(O(R_3), 1+O(R_1)) = \max(5, 1+2)$$

$$O(R_4) \qquad \overset{=5}{O(R_3)} = \max(O(R_2), 1) = \max(5,1) = 5 \qquad O(R_3) \quad O(R_1) = 2$$

$$O(R_2) = \max(O(R_1), 5) = \max(2,5) = 5$$

More precise:

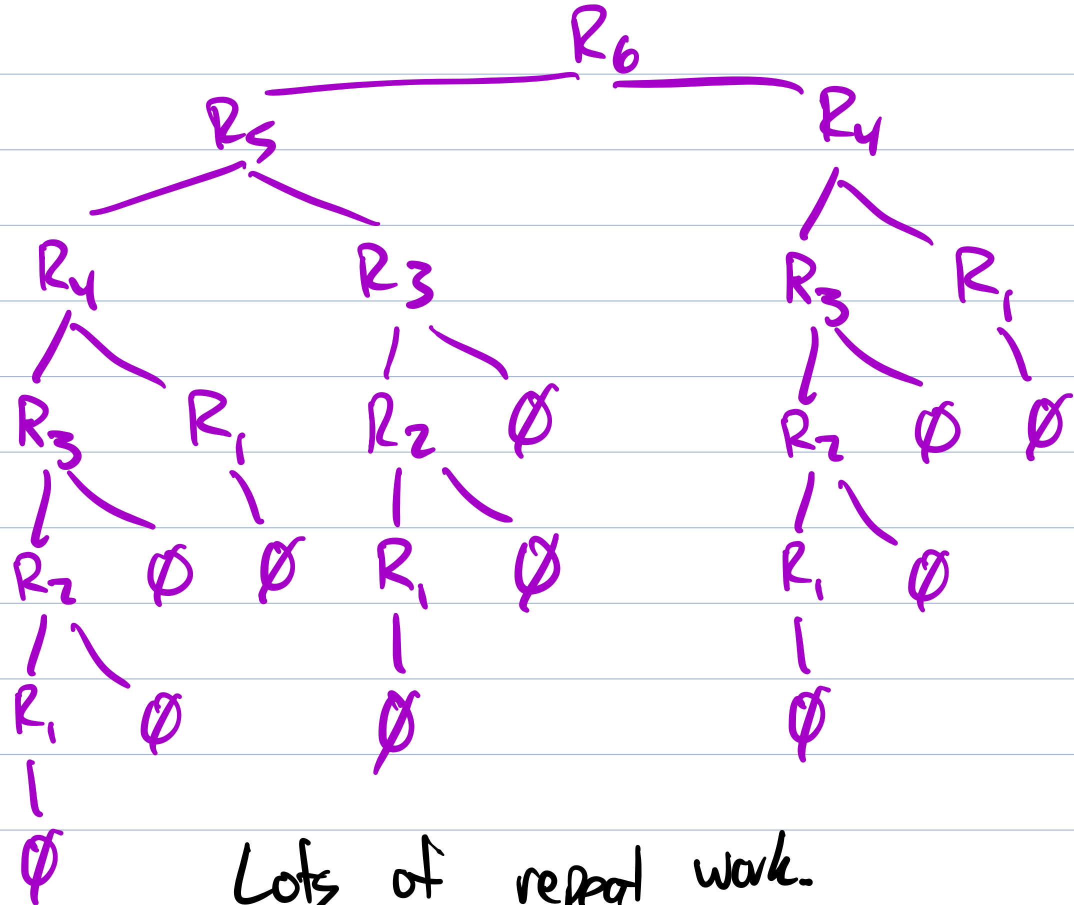
Let $p(S)$ be the intervals in S that don't conflict with the last element in S.

Ex: $p(R_6) = \{r_1, r_2, r_3, r_4\}$

$p(R_4) = \{r_1\}$

Then:
$$O(R_k) = \begin{cases} 0, & \text{if } R_k = \{\} \\ \max(O(R_{k-1}), v_k + O(p(R_k))), & R_k \neq \{\} \end{cases}$$

Implemented in code like this, it will still be exponential time because of lots of repeat work.



Lots of repeat work.

Easy to remove duplicate work if the first time we solve a case (ex: $O(R_3)$), we store the value. This is called memoization.

Pseudocode:

```
memo = empty dictionary                 sorted by end-time
function wi(S):     # should return score of best sol
    if S is a key in memo:
        return memo[S]
    if S = {}:
        memo[S] = 0
        return 0
    r = last request in S
```

v = value of r
$score = \max(wi(S - \{r\}), v + wi(p(S)))$
$memo[S] = score$
return score

This time is linear (not counting sorting!)

What does "memo" look like at the end?

$\emptyset : 0$
$R_1 : 2$
$R_2 : 5$
$R_3 : 5$
$R_4 : 5$
$R_5 : 9$
$R_6 : 12$

Once we know the best score is 12, how do we reconstruct the actual solution that has a score of 12?

$12 = wi(R_6) = \max(9, 12)$ taking r_6 and then $wi(R_4)$

$5 = wi(R_4) = \max(5, 3)$ not taking r_4, calling $wi(R_3)$

$5 = wi(R_3) = \max(5, 1)$ not taking r_3, calling $wi(R_2)$

$5 = wi(R_2) = \max(2, 5)$ taking $\boxed{r_2}$ and calling $wi(\emptyset)$

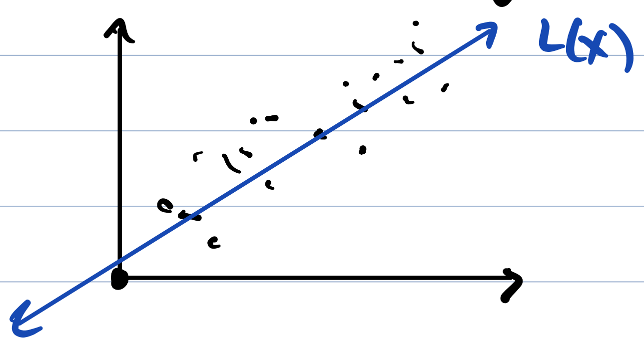
$wi(\emptyset) = 0$

Best solution $\{r_2, r_6\}$.

Bonus Topic - Dynamic Programming - part 2

Example #2 - Segmented Least Squares

Normal Linear Regression

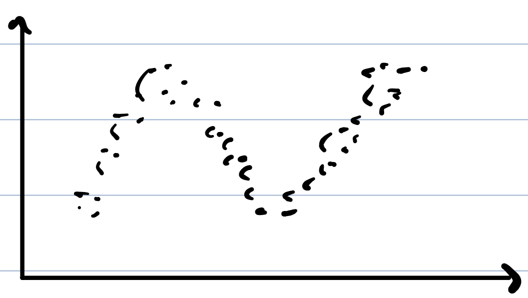


points: $(x_1, y_1), \dots, (x_n, y_n)$

goal: minimize the quantity

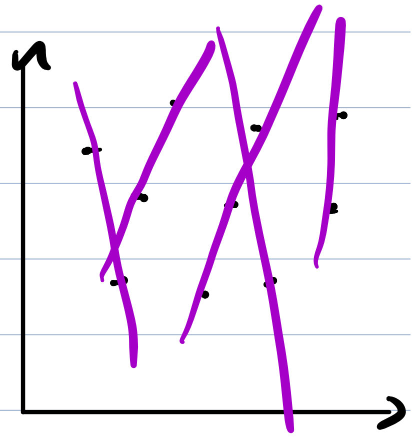
$$\sum_{i=1}^{n} \left(L(x_i) - y_i\right)^2$$

Bad for a data set like this:



Segmented Least Squares

- Split the data points into k consecutive blocks
- Find the least squares line separately for each block
- Minimize... some combo of # of lines and the errors of each line
  (we want to penalize each new line by a little)



Score:

$C \cdot (\text{\# of lines}) +$ sum of the errors of each line

$C > 0$ is a constant that we can tweak to help us find desirable solutions

Search Space: all ways of splitting the points (sorted in increasing order by x-value) into any # of non-empty consecutive blocks.

Ex: $n = 10$

$$\underbrace{1 \quad 2 \quad 3 \quad 4}\ \underbrace{5 \quad 6}\ \underbrace{7 \quad 8 \quad 9}\ \underbrace{10}$$

4 blocks

$5 + 2 + 3 = 10$

$4 + 2 + 3 + 1 = 10$

$2 + 4 + 3 + 1 = 10$

These are called "integer compositions"

Fact: # of integer compositions of n is
$$2^{n-1}$$

Solving with Dynamic Programming

Each optimal solution to this problem consists of:

- The final block $p_i, p_{i+1}, \dots, p_n$
- An optimal solution on all other points $p_1, p_2, \dots, p_{i-1}$

$$\underbrace{1\ \ 2\ \ 3\ \ 4}\ \ \underbrace{5\ \ 6}\ \ \underbrace{7\ \ 8\ \ 9}\ \ \cancel{10}$$

4 blocks

$$\underbrace{1\ \ 2\ \ 3}\ \ \underbrace{4\ \ 5}\ \ \underbrace{6\ \ 7\ \ 8\ \ 9}\ \ \underbrace{10}$$

$$\underbrace{1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9}_{\text{optimal}}\ \ \underbrace{10}$$

$$\underbrace{1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8}_{\text{optimal}}\ \ \underbrace{9\ \ 10}$$

Suppose:

* $e_{i,j}$ is the error of the minimum least squares line on the points $p_i, p_{i+1}, \ldots, p_j$.

* $O(i)$ = the optimal (minimal) score on the points $p_1, p_2, \ldots, p_i$

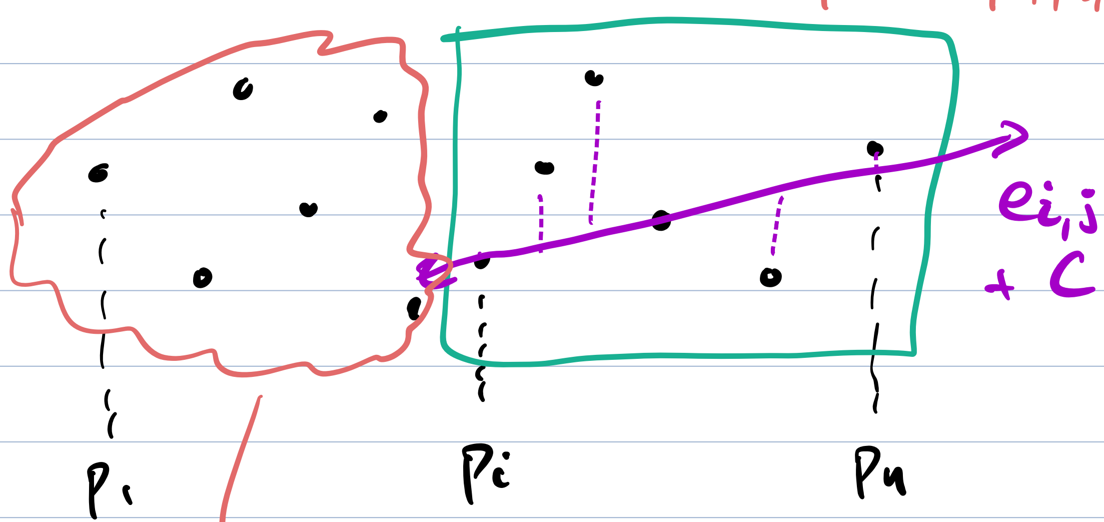
What is the score of the best solution that has last block go $p_i, p_{i+1}, \ldots, p_n$?

$$\text{cost} = e_{i,n} + C + O(i-1)$$

error of last line

penalty of last line

optimal cost for the points $p_1, p_2, \ldots, p_{i-1}$



$O(i-1)$ = best score for $p_1, p_2, \ldots, p_{i-1}$

What is the optimal score for $p_1, \ldots, p_n$?
We need to pick the value of i that minimizes

$$e_{i,n} + C + O(i-1)$$

Recurrence:

$$\begin{cases} O(0) = 0 \\ O(j) = \min\limits_{1 \le i \le j} \left(e_{i,j} + C + O(i-1) \right) \end{cases}$$

To do this in code, we can actually build from the bottom up $(O(0), O(1), O(2), \ldots)$.
This is <u>iterative</u>.

```
memo = dict()
O(0) memo[0] = 0
compute e_{i,j} for all pairs 1 ≤ i ≤ j ≤ n
for j = 1, ..., n:
O(j)= memo[j] = min(e_{i,j} + C + memo[i-1], i = 1, ..., j)
return memo[n]
```

<u>Runtime</u>: $O(n^2)$ (not including the computation of the $e_{i,j}$)
vs.
$O(2^n)$

The **key** to making dynamic programming work is figuring out what you need to know.

For SLS, you only need to know the optimal cost for each possible endpoint ($O(i-1)$)

The actual composition that gets you that score is irrelevant.

Bonus Topic - Dynamic Programming - part 3

Example #3

Suppose we want to add item n to a solution for the first $n-1$ items. What do we need to know?

(1) value of that solution
(2) weight of that solution

two things

↳ alternatively, the amount of remaining capacity

Consider items $I_1, \ldots, I_n$ with values $v_i > 0$, weights $w_i > 0$, capacity C.

Define $O(j, w)$ to be the optimal score on items $I_1, \ldots, I_j$ with total weight $\leq w$.

If item n is not in an optimal solution:

$$O(n, C) = O(n-1, C)$$

If item is in any optimal solution:

$$O(n, C) = v_n + O(n-1, C-w_n)$$

Recurrence:

$$O(j, w) = \begin{cases} 0, & j = 0 \\ O(j-1, w), & w_j > w \\ \max\big(O(j-1, w), \; v_j + O(j-1, w-w_j)\big), & \text{otherwise} \end{cases}$$

Pretty fast. What's getting memoized?

Memo should keep track of $O(j, w)$ for all $1 \le j \le n$, $0 \le w \le C$

n: # of items C: # of capacity

Memo dict will have $O(n \cdot C)$ entries

This will return the optimal score. How do you turn that into an optimal solution. Trace back through the memo dict.

Start by looking memo[(n, c)]

= 0? Solution is empty. {}

= memo[(n-1, c)]? Solution does not contain n^{th} item, so go look at memo[(n-1, c)] and repeat

= v_n + memo[(n-1, c-w_n)]?

Solution does contain n^{th} item so include it, then repeat this process with memo[(n-1, c-w_n)].

TSP - there is a D.P. algo for travelling salesman

Problem: it's $O(n \cdot 2^n)$.

Beats $O(n!)$

$\frac{n^n}{e^n}$ $\left(\frac{n}{e}\right)^n$