NOTE BOOK

Rokshana Ahmed

Algorithms

DATE: 28/4/22

M-COMPUTE-OPT(j): MEMOIZATION

```
GLOBAL M
IF j A KEY OF M:
    RETURN M[j]
ELSE:
    IF j==0:
        M[j] = 0
    ELSE:
        M[j] = max(w_j + M-COMPUTE-OPT(p(j)), M-COMPUTE-OPT(j-1))
    RETURN M[j]
```

THIS ALGORITHM FILLS UP DICTIONARY M

OBS:

① Interval j is in an optimal solution O_j if

$$w_j + OPT_{p(j)} \geq OPT_{j-1}.$$

② Interval j is <u>not</u> in an optimal solution O_j if

$$OPT_{j-1} \geq w_j + OPT_{p(j)}.$$

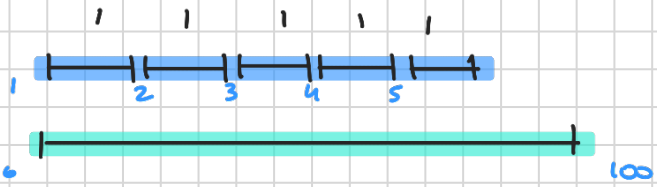
FIND SOLUTION(j): // Returns an optimal sequence of intervals among the first j

```
GLOBAL M
IF j==0:
    RETURN []
ELSE:    // should interval j be in O_j?
    IF w_j + M[p(j)] ≥ M[j-1]:
        RETURN FIND-SOLUTION(p(j)) + [j]
    ELSE:    // M[j-1] > w_j + M[p(j)]
        RETURN FIND-SOLUTION(j-1)
```

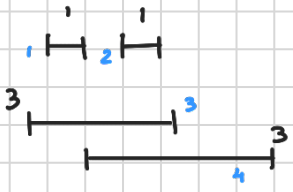
FIND-SOL(p(j)) or FIND-SOL(j-1)

T: FIND-SOLUTION(n) takes O(n) time

But, to run FIND-SOLUTION(n) we first need to fill up M (that is, to call M-COMPUTE-OPT(n))



M[0] = 0
M[1] = 1
M[2] = 2
M[3] = 3
M[4] = 4
M[5] = 5
M[6] = 100



M[0] = 0
M[1] = 1
M[2] = 2
M[3] = 3
M[4] = max(M[3], W[4] + M[1]) = max(3, 3+1) = 4

WIS(I,W): // ITERATIVE ALGO FOR FILLING UP M/OPT$_1$, ... OPT$_n$

// $I = [(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)]$, with $f_1 \leq f_2 \leq \ldots \leq f_n$

// W[i] is the weight of interval i

LET P[i] BE THE LARGEST $j < i$ S.T. $f_j < s_i$ // p(i) = P[i]

M = [None] * (n+1) O(n)

FOR i=0, 1, ..., n (n+1 iteration)

IF i==0

M[i] = 0

ELSE:

M[i] = max (w[i] + M[P[i], M[i-1]) } O(1)

RETURN M // Return M[n]

EX: PROVE THAT ONE CAN COMPUTE ALL OF P[1], P[2], ..., P[n] IN O(n) TIME (PROVIDED THAT THE INTERVALS ARE SORTED)

```
DEF WIS-SOL(I, W):
    LET P[i] BE THE LARGEST j < i S.T. f_j < s_i
    M = WIS(I, W)
    O = [ ]
    i = n
    WHILE i > 0 :
      IF M[i] == W[i] + M[P[i]]:
          O.append(i)
          i = P[i]
      ELSE:        // M[i] = M[i-1]
          i = i-1
    RETURN O
```

THE ARRAY M (THE "TABLE") IS THE KEY TO THE ALGORITHM

As we said, dynamic programming solves a problem by means of the solutions to (some of) it's subproblems.

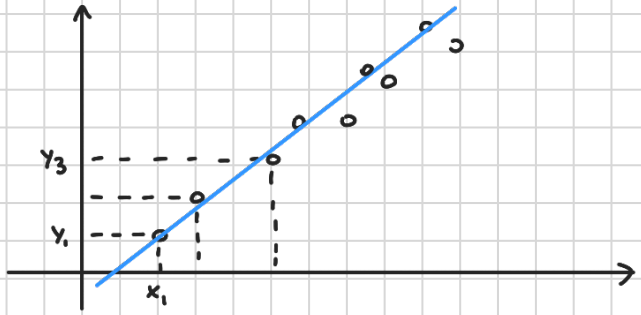
For this approach to go through, the following properties are very useful:

1. There is only a polynomial number of subproblems to consider;
2. The solution to the problem can be computed efficiently given the solutions to the subproblems;
3. Subproblems have to have some "ordering" (e.g., from smallest to largest), and there must exists a recurrence that gives us the optimal value of a problem using the optimal values of the problems preceding it in the ordering.

(THINK OF THESE AS "GUIDELINES", NOT RULES)

IN WIS, we had [2] subproblems per problem

LEAST SQUARES FIT



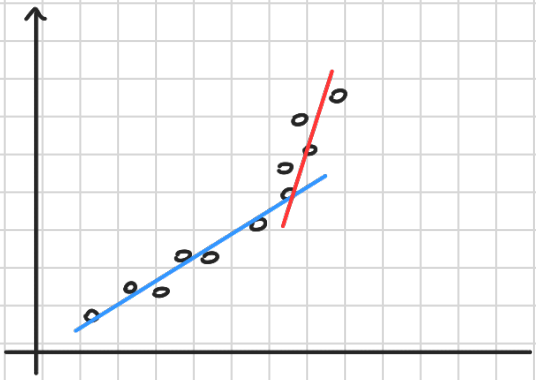
Let the set of points $P = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ be given, with $x_1 < x_2 < x_3 < \dots < x_n$

Given a line L, $y = ax + b$, we say that the error of L with respect to P is the sum of the squared distances.

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

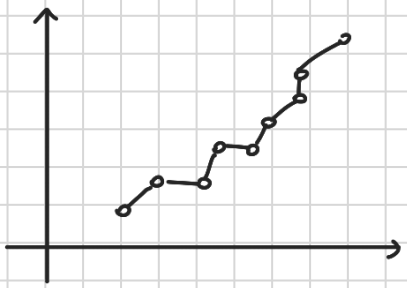
The line having the smallest error can be shown to be $y = ax + b$, with

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



Essentially, here, we would like to have two different lines (because they "look right").

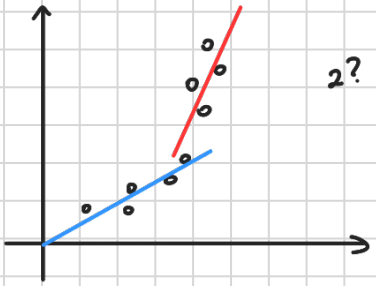
Can we, say, let the algorithm decide the optimal number of lines?

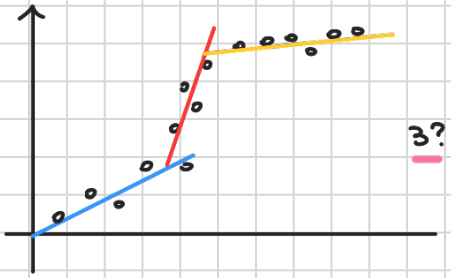


No!

For otherwise the algo might overfit (it would choose n-1 lines)

CAN WE THEN ENFORCE SOME NUMBER OF LINES?





We need to formulate the problem so that

1. the fit is good
2. the lines are few.

We have $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ with $x_1 < x_2 < \ldots < x_n$

Let $p_i = (x_i, y_i)$ For $i = 1 \ldots n$

First, we aim to partition P into "segments"
- a segment is just a continous set of points, e.g., $\{p_i, p_{i+1}, \ldots, p_n\}$ For $i \leq j$.

For each segment, we compute the optimal line as above (the line inducing the smallest error of the points of the segment).

The penalty of a partition, is the sum of the following quantities:

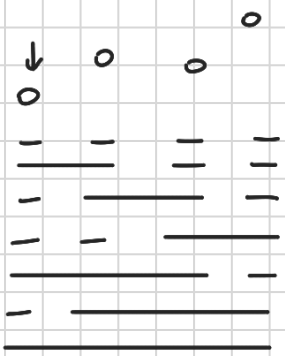
1. c times the number of segments of our partition,
2. For each segment, the error of the segment's approximating line

The goal is to find a partition of minimum penalty.

By changing "C" we can either:

- get solutions with few lines / segments ($C \to \infty$);
- get small total error ($C \to 0$)

There are exponentially many partitions.

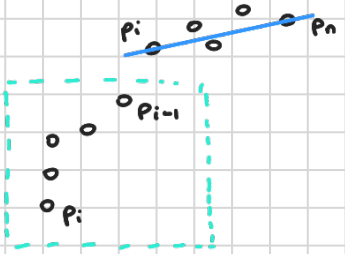


Dynamic Programming

OBSERVATION:

In the optimal solution point p_n (the last point) belongs to some segment $p_i, p_{i+1}, \dots, p_n$.

If we know what the segment is (i.e., if we know i), we can compute the cost of the segment $p_i, \dots, p_n$ (by using the above formula) and add this COST to the optimal cost for points $p_1, p_2, \dots, p_{i-1}$.



LET $OPT(i)$ BE THE OPTIMAL COST FOR POINTS $p_1, \dots, p_i$.

LET $OPT(0) = 0$

LET e_{ij} be THE MINIMUM ERROR A LINE CAN MAKE ON THE SEGMENT $p_i, p_{i+1}, \dots, p_j$. (e_{ij} CAN BE COMPUTED BY THE PREVIOUS FORMULA).

Our observation entails the following:

L: If the last segment of the optimal solution is $p_i, \dots, p_n$, then $OPT(n) = e_{i,n} + C + OPT(i-1)$

Thus,

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

The segment $p_i, \dots, p_j$ is used in an optimal solution IFF $OPT(j) = e_{i,j} + C + OPT(i-1)$.

SEGMENTED - LEAST- SQUARES (P):

$M[0] \leftarrow 0$

FOR $i = 1 \dots n$:

FOR $j = i \dots n$:

COMPUTE THE ERROR $e_{i,j}$ FOR THE SEGMENT $p_i, p_{i+1}, \dots, p_j$

FOR $j = 1 \dots n$:

$$M[j] = \min_{1 \le i \le j} (e_{i,j} + C + M[i-1])$$

RETURN $M[n]$

This returns the minimum cost.

Ex: Write an algorithm for finding the actual segmentation.

Ex: Try to make the algorithms as fast as possible.