Dynamic Programming: Iteration Over Subproblems

- Often neither greedy nor divide & conquer work

Weighted Interval Scheduling

- Interval scheduling but intervals have weights
  - Greedy does not work!

  \[ \begin{align*}
  &\text{Value} = 1 \\
  &\text{Value} = 3 \\
  &\text{Value} = 1 \\
  \end{align*} \]

  \[ \text{Total Value} = 2 \]

- Goal: Two intervals are compatible if they do not overlap.

  Select a subset \( S \subseteq \{1,\ldots,n\} \) such that

  \[ \sum_{i \in S} v_i = \text{maximum} \]

  where \( v_i \) = value of \( i \).

- Assume the requests are sorted in order of non-decreasing finish time:

  \[ f_1 \leq f_2 \leq \ldots \leq f_n \]

  i.e., \( i < j \Rightarrow f_i \leq f_j \)
Definition

\[ p(j) = \text{largest } i \text{ such that } i \leq j \text{ and the intervals } i \text{ and } j \text{ are disjoint.} \]

\[ p(j) = 0 \text{ if no request } i \leq j \text{ is disjoint from } j. \]

1  \[ \sigma_1 = 2 \]  \[ p(1) = 0 \]

2  \[ \sigma_2 = 4 \]  \[ p(2) = 0 \]

3  \[ \sigma_3 = 4 \]  \[ p(3) = 1 \]

4  \[ \sigma_4 = 7 \]  \[ p(4) = 0 \]

5  \[ \sigma_5 = 2 \]  \[ p(5) = 3 \]

6  \[ \sigma_6 = 1 \]  \[ p(6) = 3 \]
Suppose we have an optimal solution $O$.

* Case 1. $n \notin O$ \[ one \text{ of them always happens!} \]
* Case 2. $n \notin O$

1. $n \notin O$ $\Rightarrow$ no interval $p(n) < i < n$ belongs to $O$

Moreover, $O$ must include an optimal solution

to the problem of requests $\{1, \ldots, p(n)\}$.

2. $n \notin O$; $O$ is the optimal solution to the problem of requests $\{1, \ldots, n-1\}$

\[ \text{OPT}(j) \text{ - optimal solution to } \{1, \ldots, j\}, \]

and \( \text{OPT}(0) = 0 \)

\[ O_j \text{ - optimal solution to } \{1, \ldots, j\}, \]

we are looking for $O_n$ and $\text{OPT}(n)$
\[ \text{OPT}(j) = \begin{cases} v_j + \text{OPT}(p(j)) & \text{if } j \in \mathcal{O}_j \\ \text{OPT}(j-1) & \text{if } j \notin \mathcal{O} \end{cases} \]

\[ \text{OPT}(j) = \max (v_j + \text{OPT}(p(j)), \text{OPT}(j-1)) \]

**FACT:** \( j \in \mathcal{O}_j \iff v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1) \)

**ALGORITHM:**

1. Compute -Opt(j)
2. If \( j = 0 \) then Return 0
3. Else
   - Return \( \max (v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
4. End if

This may lead to computing many \( \text{Compute-Opt}(k) \) many times!
Proposition

Compute $\text{Opt}(j)$ correctly computes $\text{OPT}(j)$, for $j = 1, 2, \ldots, n$.

Proof: By induction.

Complexity

Case $p(j) = j - 2$ for $j = 2, 3, 4, \ldots$

Then

$$T(n) = T(n-1) + T(n-2) + c$$

$$T(n) \leq 2T(n-2) + c \leq 2(2T(n-4) + c)$$

$$= 2^2 T(n-4) + 2^1 c$$

$$= 2^{\frac{n}{2}} c + 2 \cdot 2^\frac{n}{2}$$

i.e.

$$T(n) = \mathcal{O}(2^n)$$
SOLUTION: MEMOIZING THE RECURSION

Idea: do not solve anything twice, store and use the first solution!

- We need some memory for storage subsolutions.
- Dynamic programming always use some arrays
- \( M[0..n] \),
  \( M[j] \) is initially "empty", but later contains \( \text{Compute-Opt}(j) \)

\[
\text{M-Compute-Opt}(j) \\
\text{If } j = 0 \text{ then Return } 0 \\
\text{Else if } M[j] \neq \emptyset \text{ then Return } M[j] \\
\text{Else} \\
M[j] = \max (v_j + \text{M-Compute}(p(j)), \text{M-Compute-Opt}(j-1)) \\
\text{Return } M[j] \\
\text{End if}
\]
Proposition

If the input intervals are sorted by their finish time, the running time of $M$-Compute-Opt($n$) is $O(n)$.

Proof:
Let NotEmpty be the number of $M[i] \neq \emptyset$.
Initially NotEmpty = 0, but in each iteration
NotEmpty $\leftarrow$ NotEmpty + 1.
But $M$ has only $n+1$ elements, so $O(n)$. \qed
Computing $C_n$

Find-Solution ($j$)

If $j=0$ then output nothing
Else
  If $j + M[p(j)] \geq M[j-1]$ then
    Output $j$ together with the result of Find-Solution ($p(j)$)
  Else
    Output the result of Find-Solution ($j-1$)
  End if
End if

Complexity:

$$T(n) = T(f(n)) + c \leq T(n-1) + c$$
$$= T(n-2) + c + c = \cdots = n \cdot c + T(0) = O(n)$$

Important!

$$T(n) = T(n-1) + c = O(n)$$
$$T(n) = 2 \cdot T(n-1) + c = O(2^n)$$
$$T(n) = 2 \cdot T(n-2) + c = O(2^n)$$

$\uparrow$ any constant!
• TOTAL COMPLEXITY: $O(n \log n)$

BECAUSE OF INITIAL SORTING!
SEGMENTED LEAST SQUARES

\[ \text{error} = \sum_{i=1}^{n} (y_i - a x_i - b)^2 = \min \]

\[ y = a x + b \]

where

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

\[ b = \frac{\sum y_i - a \sum x_i}{n} \]
which set of lines is the best approximation!
FORMULATING THE PROBLEM

- We are given a set of points \( P = \{(x_1, y_1), \ldots, (x_n, y_n)\} \) with \( x_1 < x_2 < \ldots < x_n \).
- \( P_i \) denotes \( (x_i, y_i) \).
- A segment is a subset of \( P \) of the form \( \{P_i, P_{i+1}, \ldots, P_j\} \) for \( i \leq j \).
- For each segment \( S \) we compute the line in the classical way.

- The penalty of a partition is defined by a sum of the following terms.

  1. Segmentation penalty: number of segments \( \times \) \( C \), where \( C \) is a fixed constant, \( C > 0 \)
  2. Error penalty: for each segment, the error value of the optimal line.
Algorithm Design

- A single line segment fits $p_i, p_{i+1}, \ldots, p_n$, and then an optimal solution is found for the remaining points $p_i, \ldots, p_{i-1}$.

- $\text{OPT}(i)$ - optimum solution for $p_i, \ldots, p_i$, $\text{OPT}(0) = 0$

- $e_{i,j}$ - minimum error of any line that approximate $p_i, p_{i+1}, \ldots, p_j$

\[\text{FACT} \quad \text{If the last segment of the optimal partition is } p_i, \ldots, p_n, \text{ then the value of the optimal solution is } \text{OPT}(n) = e_{i,n} + C + \text{OPT}(i-1).\]
**PROPOSITION**

For the subproblem on the points \( P_i, \ldots, P_j \)

\[
 \text{OPT}(j) = \min_{1 \leq i \leq j} (e_{i,j} + C + \text{OPT}(i-1)),
\]

and the segment \( P_i, \ldots, P_j \) is used in an optimal solution for the subproblem if and only if the minimum is obtained using index \( i \).

---

**Segmented Least-Square (n)**

*Array M[0..n]*

*Set M[0] = 0*

*For all pairs \( i \leq j \)*

  *Compute the least squares error \( e_{i,j} \) for the segment \( P_i, \ldots, P_j \)*

*Endfor*

*For \( j = 1, 2, \ldots, n \)*

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

*Endfor*

*Return M[n]*
Find-segments (j)

If $j = 0$ then Output nothing

Else

Find an $i$ that minimizes $e_{ij} + C + M[i-1]$,

Output the segment $\{p_i, ..., p_j\}$ and the result of Find-segments (i-1)

End if.

Complexity

1. We need to calculate $e_{ij}$ first.
   Calculation of one $e_{ij}$ is $O(n)$.
   We have $n^2$ of $e_{ij}$ so $O(n^3)$.
   It can be done using some tricks in $O(n^2)$ see the textbook.

2. Computing each $M[i,j]$ is $O(n)$, so $O(n^2)$.

Total $O(n^3)$ or $O(n^2)$.