Dynamic Programming

Recall Fibonacci

**Recursive**

\[ f(n) \]
\[ \begin{align*}
  &\text{if } n = 0 \text{ return } 0 \\
  &\text{if } n = 1 \text{ return } 1 \\
  &\text{else return } \\
  &\quad f(n-1) + f(n-2)
\end{align*} \]

\[ T(n) = T(n-1) + T(n-2) + c \]

so run time grows like the Fibonacci numbers

BAD

**Iterative**

\[ f(0) = 0 \]
\[ f(1) = 1 \]
\[ \text{for } i = 2 \ldots n \]
\[ f(i) = f(i-1) + f(i-2) \]

\[ O(n) \text{ (assuming numbers are small enough)} \]

GOOD

- an example of dynamic programming.

Main idea of dynamic programming:

solve "subproblems" from smaller to larger (bottom up) storing solutions

Run time:

\[ (#\text{subproblems}) \times (\text{time to solve one subproblem}) \]
Weighted Interval Scheduling

Recall Interval Scheduling aka Activity Selection:
Given a set of intervals $I$, find a max size subset of disjoint intervals

Weighted Interval Scheduling - Given $I$ and weight $w(i)$ for each $i \in I$, find set $S \subseteq I$ s.t. no two intervals in $S$ overlap and maximize $\sum_{i \in S} w(i)$

e.g. you have preferences for certain activities.

Recall greedy alg.
Order intervals $1, 2, \ldots, n$ by right endpoint

$S \leftarrow \emptyset$

for $i = 1, \ldots, n$

if interval $i$ is disjoint from those in $S$

then $S \leftarrow S \cup \{i\}$

end

This does not work for the weighted version

e.g.,

\[
\begin{array}{ccc}
1 & 5 \\
1 & & 1 \\
\end{array}
\]

What about greedy taking max weight first? No

Notation $\text{OPT}(I)$ - optimum set $S$

$w_{\text{OPT}}(I)$ - its weight
A general approach to finding $\text{OPT}(I)$:
Consider one interval $i$. Either it is in $\text{OPT}(I)$ or not.
If $i \in \text{OPT}(I)$ then $\text{OPT}(I) = \{i\} \cup \text{OPT}(I'\setminus\{i\})$
where $I' = \text{intervals disjoint from } i$
If $i \notin \text{OPT}(I)$ then $\text{OPT}(I) = \text{OPT}(I \setminus \{i\})$
We want the max of these two possibilities

$$\text{w}_{\text{OPT}}(I) = \max \{\text{w}_{\text{OPT}}(I \setminus \{i\}), \text{w}(i) + \text{w}_{\text{OPT}}(I')\}$$

In general this does not give poly time

$$T(n) = 2T(n-1) + O(1) \quad \text{— exponential.}$$

Essentially, we may end up solving subproblems for each of the $2^n$ subsets of $I$.
However, if we order intervals $1..n$ by right endpt something nice happens

Before $i$ and $i-1$

Intervals disjoint from interval $i$ are $1..j$ for some $j = p(i)$

Then $p(i) =$ largest index $k < i$ s.t. interval $k$ is disjoint from interval $i$

Let $M(i) = \text{w}_{\text{OPT}}(\{1, 2, \ldots, i\})$
Then \( M(i) = \max\{M(i-1), w(i) + M(p(i))\} \)

How to compute \( p(i) \):

we use sorted order 1...n by right endpoint
AND sorted order \( l_1,...,l_n \) by left endpoint

\[
j \leftarrow n \\
\text{for } k = n \ldots 1 \\
\text{while } l_k \text{ overlaps } j \text{ do } j \leftarrow j-1 \\
p(l_k) \leftarrow j \\
\text{end}
\]

Runtime \( \Theta(n) \) after sorting

Final algorithm

- sort intervals 1...n by right endpoint
- sort intervals by left endpoint
- compute \( p(i) \) for all \( i \)

\[
M(0) = 0 \\
\text{for } i = 1 \ldots n \\
M(i) = \max\{M(i-1), w(i) + M(p(i))\} \\
\text{end}
\]

\( M[0,...,n] \) is an array we are filling in

final answer: \( M(n) \)
Runtime $O(n \log n) + \Theta(n) + O(n \cdot c)$

$\frac{1}{\text{sort}}$ 

$\frac{1}{\text{parts}}$ 

$\text{#subproblems}$ 

$\text{time per subproblem}$

---

**How to compute the actual subset**

Recursive fix $\text{OPT}(i)$

if $i = 0$ return $\varnothing$

else if $M(i-1) \geq w(i) + M(p(i))$

return $\text{OPT}(i-1)$

else return $\{i\} \cup \text{OPT}(p(i))$

end

---

**Summary**

- A general idea to find $\text{OPT}$ subset: solve subproblems where one element is in or out
  - Exponential in general; can sometimes be efficient

- Key ideas of dynamic programming
  - Identify subproblems (not too many) and an order of solving them s.t. each subproblem can be solved by combining a few previously solved subproblems
Maximum Common Subsequence

Recall pattern matching from CS 240
Given a long string $T$ and short string $P$
find occurrences of $P$ in $T$
Useful in grep, find, etc.
Also useful & given two long strings find longest common subsequence

$\text{TARMAC}$
$\text{CATAMARAN}$
Note that we can skip letters in both strings, but must

Given strings $x_1 \ldots x_m$ and $y_1 \ldots y_n$

Let $M(i, j) =$ length of longest common subsequence of $x_1 \ldots x_i$ and $y_1 \ldots y_j$

How can we solve this subproblem based on solutions to "smaller" subproblems?

Choices: match $x_i = y_j$, skip $x_i$, skip $y_j$

$M(i, j) = \max \begin{cases} 1 + M(i-1, j-1) & \text{if } x_i = y_j \\ M(i-1, j) \\ M(i, j-1) \end{cases}$

$M(i, 0) = 0$
$M(0, j) = 0$

Solve subproblems in any order with $M(i-1, j-1), M(i-1, j), M(i, j-1)$ before $M(i, j)$
Ex. Fill in the table.

Ex. Write out the pseudo-code:

\[
\text{for } i = 0 \ldots m \quad M(i, 0) \leftarrow 0 \\
\text{for } j = 0 \ldots n \quad M(0, j) \leftarrow 0 \\
\text{for } i = 1 \ldots m \\
\text{for } j = 1 \ldots n \\
M(i, j) = \max \left\{ 1 + M(i-1, j-1) \text{ if } x_i = y_j, \right. \\
\left. M(i-1, j) \right. \\
\left. M(i, j-1) \right. \\
\right. \\
\]

Note that this is a correct ordering of \( i, j \).

In fact, if \( x_i = y_j \), we can use the first choice (no need to check \( \max \) of other two choices).

Ex. prove this.
Run-time: $O(n \cdot m \cdot c)$

- # subproblems
- time to solve one subproblem (compare 3 possibilities)

To find the actual max. common subsequence:
work backwards from $M(m, n)$. Call $OPT(m, n)$.

$OPT(i, j)$

if $M(i, j) = M(i-1, j)$ then $OPT(i-1, j)$
else if $M(i, j) = M(i, j-1)$ then $OPT(i, j-1)$
else  -- we must have matched i and j
   output $i, j$
   $OPT(i-1, j-1)$

OR we can record, when we fill $M(i, j)$,
where the max came from

Next day: more sophisticated “edit” distance
between strings.
Longest increasing subsequence

\[ L = 5 \ 2 \ 9 \ 6 \ 3 \ 7 \ 4 \]
\[ S = 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 9 \]

increasing subsequence of length 3.

\[ S = \text{sort } L. \]

Claim longest increasing subsequence of \( L \) = max common subsequence of \( L \) and \( S \)

So get \( O(n^2) \) to find longest increasing subsequence (there is a more clever \( O(n \log(n)) \) algorithm. — see wikipedia)