Dynamic Programming:

Simple example: computing Fibonacci numbers:

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \]

defined by:

\[ F_0 = 0, \quad F_1 = 1, \]

and for \( n > 1 \):

\[ F_n = F_{n-1} + F_{n-2} \]

Natural algorithm:

\[ \text{Fib1}(n): \]

\[ \text{if } n = 0, \text{ return } 0 \]
\[ \text{if } n = 1, \text{ return } 1 \]
\[ \text{return } (\text{Fib1}(n-1) + \text{Fib1}(n-2)) \]

What's running time?

Look at \( T(n) = \# \text{ of steps for computing } n^{th} \text{ Fibonacci } \# \)

\[ T(0) = O(1) \]
\[ T(1) = O(1) \]

for \( n > 1 \):

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

Then \( T(n) \geq F_n \)

and \( F_n \) is HUGE

\[ F_n \approx \frac{\phi^n}{\sqrt{5}} \quad \text{where} \quad \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \quad \text{is the golden ratio} \]

So exponential-time algorithm.
Why is Fib1 so slow?

- \( F_n \)
- \( F_{n-1} \)
- \( F_{n-2} \)
- \( F_{n-3} \)
- \( F_{n-4} \)

Recomputing many times the answer to small subproblems.

Better approach: Only compute the answer to each subproblem once. To ensure this, start with smallest \( \rightarrow \) largest (bottom-up approach).

\[
\text{Fib2}(n):
\]
\[
\text{if } n = 0, \text{ return(0)}
\]
\[
\text{if } n = 1, \text{ return(1)}
\]

create an array \( F[0 \ldots n] \)

\[
F[0] = 0, \ F[1] = 1
\]

for \( i = 2 \rightarrow n 
\]

\[
F[i] = F[i-1] + F[i-2]
\]

\[
\text{return(}F[n]\text{)}
\]

Running time: \( O(1) \) per \( i \) so \( O(n) \) total time.
Dynamic Programming approach:

1) Define subproblem in words, for example:
   \[ F(i) = \text{i}^{th} \text{ Fibonacci number} \]

2) State a recurrence in terms of smaller subproblems, for example:
   \[ F(i) = F(i-1) + F(i-2) \]

3) Solve subproblems from smallest to largest.

Longest increasing subsequence:

Input: \( n \) numbers \( a_1, a_2, \ldots, a_n \)

Example: 5, 2, 8, 6, 3, 6, 9, 7

A subsequence is a subset in order.

Example: using indices 2, 4, 5, 7 gives 2, 6, 3, 9

So a subsequence is a subset \( a_{i_1}, a_{i_2}, \ldots, a_{i_k} \) where the indices satisfy:

\[ 1 \leq i_1 < i_2 < \cdots < i_k \leq n \]

(So increasing indices)

Example: \( i_1 = 2 < i_2 = 4 < i_3 = 5 < i_4 = 7 \)
A subsequence is increasing if
\[ a_{i_1} < a_{i_2} < \ldots < a_{i_k} \]

Example: \( a_2, a_5, a_6, a_7 = 2, 3, 6, 9 \)

is an increasing subsequence since \( 2 < 3 < 6 < 9 \).

Goal: Given \( a_1, \ldots, n \), find an increasing subsequence of max length.

First, let's focus on finding just the length of the longest increasing subsequence (LIS).

First step: Define the subproblem in words.

Natural idea:
\[ S(j) = \text{length of longest increasing subsequence in } a_1, \ldots, a_j \]

Goal: Compute \( S(n) \).
How to write a recurrence for $S(j)$ in terms of $S(1), S(2), ..., S(j-1)$?

For example, in our earlier example: 5, 2, 8, 6, 3, 6, 9, 7

$S(1) = 1$, $S(2) = 1$, $S(3) = 2$, $S(4) = 2$, $S(5) = 2$

Can we figure out $S(6)$ just from $S(1), ..., S(5)$ & $a_1, ..., a_5$?

$S(5) = 2$ but can we add 6 onto it?

If it corresponds to 2, 3 then yes

But if it corresponds to 5, 8 then no

So we need track of all possible endings

Cleaner approach:

Add extra condition to the definition of the subproblem to remember what number it ends at.
Let $L(j)$ = length of longest increasing subsequence in $a_1, ..., a_j$ which ends at $a_j$
& includes $a_j$

Goal: compute $\max_j L(j)$

$L(j) = 1 + \max_i \{L(i) : i < j, a_i < a_j\}$

means maximize $L(i)$ where the variable is $i$ & try those $i$ where $i < j$ and $a_i < a_j$

LIS(A):
input: $A = [a_1, ..., a_n]$

for $j = 1 \rightarrow n$
$\quad L(j) = 1$, prev(j) = NULL
for $i = 1 \rightarrow j - 1$
$\quad$ if $L(i) + 1 > L(j)$ & $a_i < a_j$
$\quad$ then $L(j) = L(i) + 1$
$\quad$ prev(j) = i

Let $\max = 1$
for $i = 1 \rightarrow n$
$\quad$ if $L(i) > L(\max)$ then $\max = i$

Return $L(\max)$
Running time:

- $O(1)$ per i
- $O(n)$ sized loop over i
- $O(n)$ sized loop over j
- $O(1) \times O(n) \times O(n) = O(n^2)$ total time.

How to find the actual subsequence?

- Keep track of the next to last index i that gives the max
- Then backtrack

Earlier example:

$$A = 5 2 8 6 3 6 9 7$$

$$L = 1 1 2 2 2 3 4 4$$

$$\text{Prev} = \begin{bmatrix} \text{null} & \text{null} & 1 & 1 & 2 & 5 & 6 & 6 \end{bmatrix}$$

To reconstruct $S(7)$ follow the path:

- $a_5 \leftarrow a_6 \leftarrow a_6 \leftarrow a_7$
- $2 \leftarrow 3 \leftarrow 6 \leftarrow 9$

So it's: 2, 3, 6, 9
To get the subsequence add:

\[ i = \max \\text{output}(i) \]

while \text{prev}(i) \neq \text{NULL}:

\[
\begin{align*}
& i = \text{prev}(i) \\
& \text{output}(i)
\end{align*}
\]

**Key Ideas:**

- Use prefix of input for subproblems
- Strengthen subproblem by adding extra info into it.
Earlier example: 5, 2, 8, 6, 3, 4, 9, 7

View as a graph:

Edge from $i \rightarrow j$ if $i < j$ & $a_i < a_j$

Then $L(j)$ = length of longest path ending at $a_j$

This graph is a DAG = directed acyclic graph

In topological order (all edges go left to right)

We are finding the longest path in a DAG.