Dynamic Programming I

Recursion:
To solve a problem for one input,
solve on other inputs
and combine results.

Benefits: often easy to find
Problem: usually exponential time.

How can we make it faster?

Memoization: whenever you compute
\( f(x) \) in some recursive call,
store the answer. Next time
you call \( f(x) \) just return it.

Memoization: time = \( \text{possible} \)
\( \# \text{ inputs} \), \( \text{time per call} \)

Recursion: time = \( \# \text{ paths from input} \)
\to base case
Can think of inputs as a DAG. 
A \rightarrow B if B's recursion uses A.

Fibonacci:

Recursion time = # paths = F_n \approx 1.6^n
Memorization time = (# inputs) \times (time per input) = n \times n = n^2 \text{ bit operations}

Bottom-up DP: fill from left to right, looking at each edge.

Can either **pull** or **push** along edges.

**Pull**: compute a node's value when you visit it, based on previous values.

**Push**: when you visit a computed node, update future nodes that use it.

(In Fibonacci: add your value to theirs)

Generally equivalent. Sometimes only have easy access to out- or in-edges.
Longest increasing Subsequence:

Given n numbers $A_1, \ldots, A_n$
find increasing subsequence:

$s_1 < s_2 < \ldots < s_k$ (subsequence)

$A_{s_1} < A_{s_2} < \ldots < A_{s_k}$ (increasing)

or maximum length k.

will convert to longest path on DAG
$O(n^2)$ bottom-up approach:

Suppose you build your solution $S_1, \ldots, S_n$ from left to right.

Walking a path: $(S_i, A_i) \to (S_j, A_j) \ldots$

on the DAG $(i, A_i) \to (j, A_j) \iff i < j \& A_i < A_j$

Start at $(-\infty, -\infty)$ and end at $(\infty, \infty)$

Answer = longest path in DAG (-1)

because:

Any Path is an increasing sequence
any IS is a path

$\Rightarrow$ longest path = longest IS
Memoized View

\[ f_A(i) := \text{LIS ending at } A_i \]

\[ = \max_{j < i, A_j < A_i} f_A(j) + 1 \]

Answer \(= \max_{i \in [n]} f_A(i) \)

OR (for poly time)

\( f(A) \leq \text{LIS of } A. \)

- let \( i = \arg \max A_i \)
- LIS either contains \( i \) or not

\( f(A) = \max \left( f(A \setminus A_i), \left( f(A_{i+1} \setminus A_{i-1}) + 1 \right) \right) \)

How many possible inputs?

\( \text{bar to LL} \Rightarrow n^2 \text{ options} + n \text{ time per option} \Rightarrow n^3 \)
$O(n \log n)$ Version

Given two possible starts, on $A_i, x \to A_m$:

$S_i, S_i^1, S_i^j$

$S_i, S_i^1, S_i^j$

When is one clearly superior?
When are they equivalent?

Answer: only will care about length & last value

$F_m(K) := \text{minimal last value of length- } K \text{ subsequence on } 1\ldots M$

$F_{m+1}(K) = \min (F_m(K), \begin{cases} A_{m+1} & \text{if } F_m(K-1) \leq A_{m+1} \\ A_m & \text{otherwise} \end{cases})$

$F_m: [z_1, z_2, \ldots, z_t, z_{t+1}, \ldots, z_k]$

$A_{m+1} \in (z_t, z_{t+1})$:

$F_{m+1}: [z_1, z_2, \ldots, z_t, A_{m+1}, z_{t+2}, \ldots, z_k]$

$\Rightarrow$ update is $O(n \log n)$ binary search

$\Rightarrow O(n \log^2 n)$ time
Interval Scheduling:

Want to compute $\text{Sched}(I)$
where $I = \text{set of } (s_i, f_i, w_i)$ pairs
maximize $\sum_{i \in S} w_i$
for $S \subseteq I$ non-overlapping.

Naive recursion:
$\text{Sched}(I)$
Let $i = \text{first elt of } I$
Return $\min$ of
not chosen: $\text{Sched}(I \setminus i)$
chosen: $\text{Sched}(I \setminus \hat{i} \setminus i$ or anything conflicting with $i \in S) + w_i$.

Problem: $2^n$ possible inputs.
Solution: If $I$ sorted by $f_i$, then
only $n+1$ inputs ever happen:
(suffix of $I$ sorted by $s_i$)

$\Rightarrow$ memoized time is $n \cdot (\text{time per input})$

$n^2$ naively
$n$ more carefully
$\text{Sched}($index in $I$, $f$ or last chosen$)$
DAG: sort by \( f_i \).

1 \( \rightarrow \) 2 if \( f_i \leq S_j \), or weight \( w_i \).

\( S \rightarrow \) everything weight 0

everything \( \rightarrow \) + weight \( w_j \)

Answer = \text{max weight } S \rightarrow + \text{ path.}

Path \( S \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_k \rightarrow + \)

- weight = 0 + w_1 + w_2 + \ldots + w_k
  = \text{total value of scheduling then}

- \( f_{i_1} \leq S_i \leq f_{i_2} \leq \ldots \leq f_{i_k} \)

\( \Rightarrow \) is valid schedule.

Many DP problems have this form:

Convert to a DAG
Find max-weight (or min-weight) \( S \rightarrow + \text{ path.} \)

Time = \# edges in DAG.
Stamps

values $S_1, \ldots, S_n$

with collection of value $C$, w/ fewest stamps

What is the DAG?

nodes = value

$x \rightarrow x + S_i \quad \forall i, x$, of weight 1

Answer = Shortest $0 \rightarrow C$ path.

Time = # edges = $O(N)$. 