What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

Dynamic Programming:
A recursion that when memoized leads to an efficient algorithm.

Key Questions:
- Given a recursive algorithm, how do we analyze the complexity when it is memoized?
- How do we recognize whether a problem admits a dynamic programming based efficient algorithm?
- How do we further optimize time and space of a dynamic programming based algorithm?
Part I

Edit Distance
Edit Distance

**Definition**

Edit distance between two words $X$ and $Y$ is the number of letter insertions, letter deletions and letter substitutions required to obtain $Y$ from $X$.

**Example**

The edit distance between FOOD and MONEY is at most 4:

$$\text{FOOD} \rightarrow \text{MOOD} \rightarrow \text{MONOD} \rightarrow \text{MONED} \rightarrow \text{MONEY}$$
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

\[
\begin{array}{ccccc}
F & O & O & D \\
M & O & N & E & Y
\end{array}
\]

Formally, an alignment is a sequence \( M \) of pairs \( (i, j) \) such that each index appears exactly once, and there is no “crossing”: if \( (i, j), \ldots, (i', j') \) then \( i < i' \) and \( j < j' \). One of \( i \) or \( j \) could be empty, in which case no comparison.
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

FOOD
MONEY

Formally, an alignment is a sequence $M$ of pairs $(i, j)$ such that each index appears exactly once, and there is no “crossing”: if $(i, j), ..., (i', j')$ then $i < i'$ and $j < j'$. One of $i$ or $j$ could be empty, in which case no comparison. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (, 4), (4, 5)\}$.

Cost of an alignment: the number of mismatched columns.
Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.
Edit Distance

Basic observation

Let \( X = \alpha x \) and \( Y = \beta y \)

\( \alpha, \beta \): strings. \( x \) and \( y \) single characters.

Possible alignments between \( X \) and \( Y \)

\[
\begin{array}{c|c}
\alpha & x \\
\hline
\beta & y \\
\end{array}
\quad \text{or} \quad
\begin{array}{c|c}
\alpha & x \\
\hline
\beta y & \\
\end{array}
\quad \text{or} \quad
\begin{array}{c|c}
\alpha x & \\
\hline
\beta & y \\
\end{array}
\]

Observation

Prefixes must have optimal alignment!

\[
EDIST(X, Y) = \min \left\{ EDIST(\alpha, \beta) + [x \neq y], 1 + EDIST(\alpha, Y), 1 + EDIST(X, \beta) \right\}
\]
Subproblems and Recurrence

Each subproblem corresponds to a prefix of $X$ and a prefix of $Y$

Optimal Costs

Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\text{Opt}(i, j) = \min \begin{cases} [x_i \neq y_j] + \text{Opt}(i - 1, j - 1), \\ 1 + \text{Opt}(i - 1, j), \\ 1 + \text{Opt}(i, j - 1) \end{cases}$$

Base Cases: $\text{Opt}(i, 0) = i$ and $\text{Opt}(0, j) = j$

$X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$, we wish to compute $\text{Opt}(m, n)$. 

$\text{Opt}(i, j)$ is the cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. The recurrence relation for $\text{Opt}(i, j)$ is defined as the minimum of three options: adding a penalty for a mismatch, or aligning the prefixes $x_1 \cdots x_{i-1}$ and $y_1 \cdots y_{j-1}$, or aligning the prefixes $x_1 \cdots x_i$ and $y_1 \cdots y_{j-1}$.

The base cases are $\text{Opt}(i, 0) = i$ and $\text{Opt}(0, j) = j$, which represent the costs of aligning an empty string with a string of length $i$ or $j$ respectively.
Figure: Iterative algorithm in previous slide computes values in row order.
Computing in column order to save space

Figure: $M(i, j)$ only depends on previous column values. Keep only two columns and compute in column order.
Recall

\[ M(i, j) = \min \begin{cases} 
[x_i \neq y_j] + M(i - 1, j - 1), \\
1 + M(i - 1, j), \\
1 + M(i, j - 1)
\end{cases} \]

Entries in \( j \)th column only depend on \((j - 1)\)st column and earlier entries in \( j \)th column

Only store the current column and the previous column reusing space; \( N(i, 0) \) stores \( M(i, j - 1) \) and \( N(i, 1) \) stores \( M(i, j) \)
Space Efficient Algorithm

for all \( i \) do \( N[i, 0] = i \)
for \( j = 1 \) to \( n \) do
  \( N[0, 1] = j \) (* corresponds to \( M(0, j) \) *)
for \( i = 1 \) to \( m \) do
  \( N[i, 1] = \min\left\{ [x_i \neq y_j] + N[i - 1, 0] \right. \\
  \quad \left. + N[i - 1, 1] \right\} \\
for \( i = 1 \) to \( m \) do
  Copy \( N[i, 0] = N[i, 1] \)

Analysis
Running time is \( O(mn) \) and space used is \( O(2m) = O(m) \)
Finding an Optimum Solution

The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

Can find minimum edit distance in $O(m + n)$ space and $O(mn)$ time.

**Previous Exercise:** Find an optimum alignment in $O(mn)$ space and time.
Finding an Optimum Solution

The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

Can find minimum edit distance in $O(m + n)$ space and $O(mn)$ time.

**Previous Exercise:** Find an optimum alignment in $O(mn)$ space and time.

**Today:** Finding an optimum alignment and cost in $O(m + n)$ space and $O(mn)$ time.
Divide and Conquer Approach

Fix an optimum alignment between $X[1..m]$ and $Y[1..n]$
Divide and Conquer Approach

Fix an optimum alignment between $X[1..m]$ and $Y[1..n]$

In this optimum alignment $X[1..\frac{m}{2}]$ is aligned with $Y[1..h]$ for some $h$ where $1 \leq h \leq n$. (Need not be unique but we can choose smallest such $h$). Call this $\text{Half}(X, Y)$
Divide and Conquer Approach

Fix an optimum alignment between $X_{1..m}$ and $Y_{1..n}$

In this optimum alignment $X_{1.. \frac{m}{2}}$ is aligned with $Y_{1..h}$ for some $h$ where $1 \leq h \leq n$. (Need not be unique but we can choose smallest such $h$). Call this $\text{Half}(X, Y)$

Suppose we can find $h = \text{Half}(X, Y)$ in time $O(mn)$ time and $O(m + n)$ space, that is, in the same time as finding $\text{Opt}(m, n)$ the optimum value of the alignment between $X$ and $Y$. 
Divide and Conquer Algorithm

**Linear-Space-Alignment**($X[1..m], Y[1..n]$)

If $m = 1$ use basic algorithm in $O(n)$ time and $O(n)$ space
If $n = 1$ use basic algorithm in $O(m)$ time and $O(n)$ space

Compute $h = \text{Half}(X, Y)$ in $O(mn)$ time and $O(m + n)$ space
**Linear-Space-Alignment**($X[1..m/2], Y[1..h]$)
**Linear-Space-Alignment**($X[m/2 + 1..m], Y[h + 1..n]$)
Output concatenation of the two alignments
Divide and Conquer Algorithm

\begin{algorithm}
\textbf{Linear-Space-Alignment}(X[1..m], Y[1..n])
\begin{enumerate}
\item If \( m = 1 \) use basic algorithm in \( O(n) \) time and \( O(n) \) space
\item If \( n = 1 \) use basic algorithm in \( O(m) \) time and \( O(n) \) space
\end{enumerate}

Compute \( h = \text{Half}(X, Y) \) in \( O(mn) \) time and \( O(m + n) \) space

\textbf{Linear-Space-Alignment}(X[1..m/2], Y[1..h])
\textbf{Linear-Space-Alignment}(X[m/2 + 1..m], Y[h + 1..n])

Output concatenation of the two alignments
\end{algorithm}

\textbf{Correctness:} Clear based on definition of \( \text{Half}(X, Y) \).
Divide and Conquer Algorithm

**Linear-Space-Alignment**(\(X[1..m], Y[1..n]\))

If \(m = 1\) use basic algorithm in \(O(n)\) time and \(O(n)\) space
If \(n = 1\) use basic algorithm in \(O(m)\) time and \(O(n)\) space

Compute \(h = \text{Half}(X, Y)\) in \(O(mn)\) time and \(O(m + n)\) space

**Linear-Space-Alignment**(\(X[1..m/2], Y[1..h]\))

**Linear-Space-Alignment**(\(X[m/2 + 1..m], Y[h + 1..n]\))

Output concatenation of the two alignments

**Correctness:** Clear based on definition of \(\text{Half}(X, Y)\).

**Recurrences:**

Time bound \(T(m, n) =\)
Divide and Conquer Algorithm

\textbf{Linear-Space-Alignment}(X[1..m], Y[1..n])

If \( m = 1 \) use basic algorithm in \( O(n) \) time and \( O(n) \) space
If \( n = 1 \) use basic algorithm in \( O(m) \) time and \( O(n) \) space

Compute \( h = \text{Half}(X, Y) \) in \( O(mn) \) time and \( O(m + n) \) space
\textbf{Linear-Space-Alignment}(X[1..m/2], Y[1..h])
\textbf{Linear-Space-Alignment}(X[m/2 + 1..m], Y[h + 1..n])
Output concatenation of the two alignments

\textbf{Correctness:} Clear based on definition of \( \text{Half}(X, Y) \).

\textbf{Recurrences:}
Time bound \( T(m, n) = T(m/2, h) + T(m/2, n - h) + cmn \)
Space bound \( S(m, n) = \)
Divide and Conquer Algorithm

Linear-Space-Alignment($X[1..m], Y[1..n]$)
- If $m = 1$ use basic algorithm in $O(n)$ time and $O(n)$ space
- If $n = 1$ use basic algorithm in $O(m)$ time and $O(n)$ space
- Compute $h = \text{Half}(X, Y)$ in $O(mn)$ time and $O(m + n)$ space
- Linear-Space-Alignment($X[1..m/2], Y[1..h]$)
- Linear-Space-Alignment($X[m/2 + 1..m], Y[h + 1..n]$)
- Output concatenation of the two alignments

Correctness: Clear based on definition of $\text{Half}(X, Y)$.

Recurrences:
- Time bound $T(m, n) = T(m/2, h) + T(m/2, n - h) + cmn$
- Space bound $S(m, n) = \max\{S(m/2, h), S(m/2, n - h), c(m + n)\} + O(1)$
Divide and Conquer Algorithm

**Linear-Space-Alignment**$(X[1..m], Y[1..n])$

If $m = 1$ use basic algorithm in $O(n)$ time and $O(n)$ space
If $n = 1$ use basic algorithm in $O(m)$ time and $O(n)$ space

Compute $h = \text{Half}(X, Y)$ in $O(mn)$ time and $O(m + n)$ space

**Linear-Space-Alignment**$(X[1..m/2], Y[1..h])$
**Linear-Space-Alignment**$(X[m/2 + 1..m], Y[h + 1..n])$
Output concatenation of the two alignments

**Correctness:** Clear based on definition of $\text{Half}(X, Y)$.

**Recurrences:**
Time bound $T(m, n) = T(m/2, h) + T(m/2, n - h) + cmn$
Space bound $S(m, n) = \max\{S(m/2, h), S(m/2, n - h), c(m + n)\} + O(1)$

**Claim:** $T(m, n) = O(mn)$ and $S(m, n) = O(m + n)$. 
Proof: Time bound

\[ T(m, n) \leq \begin{cases} 
  cm & \text{if } n \leq 1 \\
  cn & \text{if } m \leq 1 \\
  T(m/2, h) + T(m/2, n - h) + cmn & \text{otherwise}
\end{cases} \]
Proof: Time bound

\[ T(m, n) \leq \begin{cases} 
  cm & \text{if } n \leq 1 \\
  cn & \text{if } m \leq 1 \\
  T(m/2, h) + T(m/2, n - h) + cmn & \text{otherwise}
\end{cases} \]

Claim: \( T(m, n) \leq 2cmn \) by induction on \( m + n \).

Inductive step:

\[
T(m, n) \leq 2ch\frac{m}{2} + 2c(n - h)\frac{m}{2} + cmn \\
\leq 2cmn
\]
Proof: Space bound

\[ S(m, n) \leq \begin{cases} 
  cm & \text{if } n \leq 1 \\
  cn & \text{if } m \leq 1 \\
  \max\{S\left(\frac{m}{2}, h\right), S\left(\frac{m}{2}, n - h\right), c(m + n)\} + O(1) 
\end{cases} \]

We can reuse space for computing \textbf{Half}(X, Y). And storing the alignment can be accounted separately as \( O(m + n) \).
Proof: Space bound

\[ S(m, n) \leq \begin{cases} 
  cm & \text{if } n \leq 1 \\
  cn & \text{if } m \leq 1 \\
  \max\{S\left(\frac{m}{2}, h\right), S\left(\frac{m}{2}, n - h\right), c(m + n)\} + O(1) 
\end{cases} \]

We can reuse space for computing \textbf{Half}(X, Y). And storing the alignment can be accounted separately as \( O(m + n) \).

Claim: \( S(m, n) \leq c(m + n) + O(\log m) \).
Computing $\text{Half}(X, Y)$

Want to find $h$ such that

$$\text{EDIST}(X, Y) = \text{EDIST}(X[1..m/2], Y[1..h]) + \text{EDIST}(X[(m/2 + 1)..m], Y[(h + 1)..n])$$
Computing \textbf{Half}(X, Y)

Want to find \( h \) such that

\[
\text{EDIST}(X, Y) = \text{EDIST}(X[1..m/2], Y[1..h]) \\
+ \text{EDIST}(X[(m/2 + 1)..m], Y[(h + 1)..n])
\]

Instead compute for all \( k \) where \( 1 \leq k \leq n 
\)
(1) \( \text{EDIST}(X[1..m/2], Y[1..k]) \) &
(2) \( \text{EDIST}(X[(m/2 + 1)..m], Y[(k + 1)..n]) \)
Computing $\text{Half}(X, Y)$

Want to find $h$ such that

$$EDIST(X, Y) = EDIST(X[1..m/2], Y[1..h]) + EDIST(X[(m/2 + 1)..m], Y[(h + 1)..n])$$

Instead compute for all $k$ where $1 \leq k \leq n$,

1. $EDIST(X[1..m/2], Y[1..k])$ &
2. $EDIST(X[(m/2 + 1)..m], Y[(k + 1)..n])$

And compute $h$ as

$$\min_k(EDIST(X[1..\frac{m}{2}], Y[1..k]) + EDIST(X[(\frac{m}{2} + 1)..m], Y[(k + 1)..n]))$$
Computing $\text{Half}(X, Y)$

(1) Compute for all $1 \leq k \leq n$, $\text{EDIST}(X[1..\frac{m}{2}], Y[1..k])$
Computing $\text{Half}(X, Y)$

(1) Compute for all $1 \leq k \leq n$, $\text{EDIST}(X[1..\frac{m}{2}], Y[1..k])$

**Claim:** All values available if we compute $\text{EDIST}(X[1..\frac{m}{2}], Y[1..n])$ which we can do in $O(mn)$ time.

If $M$ is the resulting table, what entries?
Computing $\text{Half}(X, Y)$

(1) Compute for all $1 \leq k \leq n$, $\text{EDIST}(X[1..\frac{m}{2}], Y[1..k])$

**Claim:** All values available if we compute $\text{EDIST}(X[1..\frac{m}{2}], Y[1..n])$ which we can do in $O(mn)$ time.

If $M$ is the resulting table, what entries? $M(\frac{m}{2}, k)$ for all $1 \leq k \leq n$. 
Computing $\text{Half}(X, Y)$

(1) Compute for all $1 \leq k \leq n$, $\text{EDIST}(X[1..\frac{m}{2}], Y[1..k])$

Claim: All values available if we compute $\text{EDIST}(X[1..\frac{m}{2}], Y[1..n])$ which we can do in $O(mn)$ time.

If $M$ is the resulting table, what entries? $M(\frac{m}{2}, k)$ for all $1 \leq k \leq n$.

Can we do it in $O(m + n)$ space?

Yes! Use the space saving trick in computing edit distance and store the last row!
Computing $\text{Half}(X, Y)$

(2) Compute for all $1 \leq k \leq n$,
    \[ \text{EDIST}(X[\left(\frac{m}{2} + 1\right)..m], Y[(k + 1)..n]) \]
Computing $\text{Half}(X, Y)$

(2) Compute for all $1 \leq k \leq n$,
$$\text{EDIST}(X[(m/2 + 1)\ldots m], Y[(k + 1)\ldots n])$$

If we compute $\text{EDIST}(X[(m/2 + 1)\ldots m], Y[1\ldots n])$ we get the values $\text{EDIST}(X[(m/2 + 1)\ldots m], Y[1\ldots k])$ for $1 \leq k \leq n$ which is not what we quite want.
Computing $\text{Half}(X, Y)$

(2) Compute for all $1 \leq k \leq n$,
$$\text{EDIST}(X[((m/2) + 1)\ldots m], Y[(k + 1)\ldots n])$$

If we compute $\text{EDIST}(X[(m/2 + 1)\ldots m], Y[1\ldots n])$ we get the values $\text{EDIST}(X[(m/2 + 1)\ldots m], Y[1\ldots k])$ for $1 \leq k \leq n$ which is not what we quite want.

Observation: $\text{EDIST}(X, Y) = \text{EDIST}($reverse$(X), \text{reverse}(Y))$. 
Computing \textbf{Half}(X, Y)

(2) Compute for all \(1 \leq k \leq n\),
\[ \text{EDIST}(X[\left(\frac{m}{2} + 1\right)\ldots m], Y[k + 1\ldots n]) \]

If we compute \(\text{EDIST}(X[\left(\frac{m}{2} + 1\right)\ldots m], Y[1\ldots n])\) we get the values \(\text{EDIST}(X[\left(\frac{m}{2} + 1\right)\ldots m], Y[1\ldots k])\) for \(1 \leq k \leq n\) which is not what we quite want.

\textbf{Observation:} \(\text{EDIST}(X, Y) = \text{EDIST}(\text{reverse}(X), \text{reverse}(Y))\).

Hence compute \(\text{EDIST}(A, B)\) where \(A\) is reverse of \(X[\left(\frac{m}{2} + 1\right)\ldots m]\) and \(B\) is reverse of \(Y[1\ldots n]\) and this will give all the desired values.
Part II

Longest Increasing Subsequence
Sequences

Definition

**Sequence**: an ordered list \( a_1, a_2, \ldots, a_n \). **Length** of a sequence is number of elements in the list.

Definition

\( a_{i_1}, \ldots, a_{i_k} \) is a **subsequence** of \( a_1, \ldots, a_n \) if
\[ 1 \leq i_1 < i_2 < \ldots < i_k \leq n. \]

Definition

A sequence is **increasing** if \( a_1 < a_2 < \ldots < a_n \). It is **non-decreasing** if \( a_1 \leq a_2 \leq \ldots \leq a_n \). Similarly **decreasing** and **non-increasing**.
Sequences

Example...

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1, 9, 1
2. Subsequence of above sequence: 5, 2, 1
3. Increasing sequence: 3, 5, 9, 17, 54
4. Decreasing sequence: 34, 21, 7, 5, 1
5. Increasing subsequence of the first sequence: 2, 7, 9.
Longest Increasing Subsequence Problem

**Input**  A sequence of numbers $a_1, a_2, \ldots, a_n$

**Goal**  Find an *increasing subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length
Longest Increasing Subsequence Problem

Input  A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal   Find an increasing subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8
Recursive Algorithm

**Definition**

\[ \text{LISEnding}(A[1..k]) : \text{length of longest increasing sub-sequence that ends in } A[k]. \]

**Question:** can we obtain a recursive expression?
Recursive Algorithm

**Definition**

\[ \text{LISEnding}(A[1..k]) : \text{length of longest increasing sub-sequence that ends in } A[k]. \]

**Question:** can we obtain a recursive expression?

\[ \text{LISEnding}(A[1..k]) = 1 + \max_{i: A[i] < A[k]} \text{LISEnding}(A[1..i]) \]
**Recursive Algorithm**

**Definition**

\[ \text{LISEnding}(A[1..k]) : \text{length of longest increasing sub-sequence that ends in } A[k]. \]

**Question:** can we obtain a recursive expression?

\[
\text{LISEnding}(A[1..k]) = 1 + \max_{i : A[i] < A[k]} \text{LISEnding}(A[1..i])
\]
Recursive Algorithm

Definition

\( \text{LISEnding}(A[1..k]) \): length of longest increasing sub-sequence that ends in \( A[k] \).

**Question:** can we obtain a recursive expression?

\[
\text{LISEnding}(A[1..k]) = 1 + \max_{i: A[i] < A[k]} \text{LISEnding}(A[1..i])
\]

\[
\text{LISEnding}(A[1..k]) = \max_{i: A[i] < A[k]} (1 + \text{LISEnding}(A[1..i]))
\]
Example

Sequence:  \( A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9 \)
Recursive Algorithm

\[
\text{LIS\_ending\_alg}(A[1..k]):
\]
\[
\text{if } (k = 0) \text{ return } 0
\]
\[
m = 1
\]
\[
\text{for } i = 1 \text{ to } k - 1 \text{ do}
\]
\[
\text{if } (A[i] < A[k]) \text{ then}
\]
\[
m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i]))
\]
\[
\text{return } m
\]

\[
\text{LIS}(A[1..n]):
\]
\[
\text{return } \max_{k=1}^{n} \text{LIS\_ending\_alg}(A[1 \ldots k])
\]
Recursive Algorithm

\begin{algorithmic}
\ifdef{LIS\_ending\_alg}{LIS\_ending\_alg}{\textbf{LIS\_ending\_alg}}(A[1..k]):
  \textbf{if} (k = 0) \textbf{return} 0
  \textit{m} = 1
  \textbf{for} \textit{i} = 1 \textbf{to} k - 1 \textbf{do}
    \textbf{if} (A[i] < A[k]) \textbf{then}
      \textit{m} = \max(\textit{m}, 1 + \textbf{LIS\_ending\_alg}(A[1..i]))
  \textbf{return} \textit{m}
\end{algorithmic}

\begin{algorithmic}
\ifdef{LIS}{LIS}{\textbf{LIS}}(A[1..n]):
  \textbf{return} \max_{k=1}^{n} \textbf{LIS\_ending\_alg}(A[1 \ldots k])
\end{algorithmic}

How many distinct sub-problems will \textbf{LIS\_ending\_alg}(A[1..n]) generate?
Recursive Algorithm

**LIS\_ending\_alg\((A[1..k])\):**
- if \((k = 0)\) return 0
- \(m = 1\)
- for \(i = 1\) to \(k - 1\) do
  - if \((A[i] < A[k])\) then
    - \(m = \max(m, 1 + LIS\_ending\_alg(A[1..i]))\)
- return \(m\)

**LIS\((A[1..n])\):**
- return \(\max_{k=1}^{n} LIS\_ending\_alg(A[1\ldots k])\)

- How many distinct sub-problems will \(LIS\_ending\_alg(A[1..n])\)
generate? \(O(n)\)
Recursive Algorithm

LIS_ending_alg(A[1..k]):
    if (k = 0) return 0
    m = 1
    for i = 1 to k − 1 do
        if (A[i] < A[k]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m

LIS(A[1..n]):
    return max^n_k=1 LIS_ending_alg(A[1...k])

- How many distinct sub-problems will LIS_ending_alg(A[1..n]) generate? $O(n)$
- What is the running time if we memoize recursion?
Recursive Algorithm

\[
\text{LIS\_ending\_alg}(A[1..k]) :
\begin{align*}
&\text{if } (k = 0) \text{ return } 0 \\
&m = 1 \\
&\text{for } i = 1 \text{ to } k - 1 \text{ do} \\
&\quad \text{if } (A[i] < A[k]) \text{ then} \\
&\quad \quad m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i])) \\
&\text{return } m
\end{align*}
\]

\[
\text{LIS}(A[1..n]) :
\begin{align*}
&\text{return } \max_{k=1}^{n} \text{LIS\_ending\_alg}(A[1...k])
\end{align*}
\]

- How many distinct sub-problems will \text{LIS\_ending\_alg}(A[1..n]) generate? \(O(n)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(n)\) time
Recursive Algorithm

\textbf{LIS\textunderscore ending\textunderscore alg}(A[1..k]):
\begin{align*}
\text{if } (k = 0) \text{ return } 0 \\
m = 1 \\
\text{for } i = 1 \text{ to } k - 1 \text{ do} \\
\quad \text{if } (A[i] < A[k]) \text{ then} \\
\qquad m = \max(m, 1 + \text{LIS\textunderscore ending\textunderscore alg}(A[1..i])) \\
\text{return } m
\end{align*}

\textbf{LIS}(A[1..n]):
\begin{align*}
\text{return } \max_{k=1}^{n} \text{LIS\textunderscore ending\textunderscore alg}(A[1 \ldots k])
\end{align*}

- How many distinct sub-problems will \textbf{LIS\textunderscore ending\textunderscore alg}(A[1..n]) generate? \(O(n)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(n)\) time
- How much space for memoization?
Recursive Algorithm

\[
\text{LIS}_\text{ending}\_\text{alg}(A[1..k]): \\
\quad \text{if } (k = 0) \text{ return } 0 \\
\quad m = 1 \\
\quad \text{for } i = 1 \text{ to } k - 1 \text{ do} \\
\quad \quad \text{if } (A[i] < A[k]) \text{ then} \\
\quad \quad \quad m = \max(m, 1 + \text{LIS}_\text{ending}\_\text{alg}(A[1..i])) \\
\quad \text{return } m
\]

\[
\text{LIS}(A[1..n]): \\
\quad \text{return } \max_{k=1}^{n} \text{LIS}_\text{ending}\_\text{alg}(A[1 \ldots k])
\]

- How many distinct sub-problems will \text{LIS}_\text{ending}\_\text{alg}(A[1..n]) generate? \(O(n)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(n)\) time
- How much space for memoization? \(O(n)\)
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why?
Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an iterative algorithm via explicit memoization and bottom up computation.

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.
Iterative Algorithm via Memoization

Compute the values \( \text{LIS\_ending\_alg}(A[1..i]) \) iteratively in a bottom up fashion.

\[
\begin{align*}
\text{LIS\_ending\_alg}(A[1..n]): \\
& \quad \text{Array } L[1..n] \quad (* L[k] = \text{value of LIS\_ending\_alg}(A[1..k]) *) \\
& \quad \text{for } k = 1 \text{ to } n \text{ do} \\
& \quad \quad L[k] = 1 \\
& \quad \quad \text{for } i = 1 \text{ to } k - 1 \text{ do} \\
& \quad \quad \quad \quad \text{if } (A[i] < A[k]) \text{ do} \\
& \quad \quad \quad \quad \quad L[k] = \max(L[k], 1 + L[i]) \\
& \quad \text{return } L
\end{align*}
\]

\[
\begin{align*}
\text{LIS}(A[1..n]): \\
& \quad L = \text{LIS\_ending\_alg}(A[1..n]) \\
& \quad \text{return} \text{ the maximum value in } L
\end{align*}
\]
Iterative Algorithm via Memoization

Simplifying:

\[
\text{LIS}(A[1..n]): \quad \text{Array } L[1..n] \quad (* \text{ } L[k] \text{ stores } \text{LISEnding}(A[1..k]) *) \\
\text{m} = 0 \\
\text{for } k = 1 \text{ to } n \text{ do} \\
\quad L[k] = 1 \\
\quad \text{for } i = 1 \text{ to } k - 1 \text{ do} \\
\quad \quad \text{if } (A[i] < A[k]) \text{ do} \\
\quad \quad \quad L[k] = \max(L[k], 1 + L[i]) \\
\quad \quad m = \max(m, L[k]) \\
\text{return } m
\]

Correctness: Via induction following the recursion
Running time: \(O(n^2)\)
Space: \(\Theta(n)\)
Improving run time

Want to improve run time to $O(n \log n)$ from $O(n^2)$. How?

**Idea:** Use data structures to improve run-time of computing

$$\text{LISEnding}(k) = \max_{i < k : A[i] < A[k]} 1 + \text{LISEnding}(i)$$
Improving run time

Want to improve run time to $O(n \log n)$ from $O(n^2)$. How?

**Idea:** Use data structures to improve run-time of computing

$$\text{LISEnding}(k) = \max_{i < k: A[i] < A[k]} 1 + \text{LISEnding}(i)$$

- When computing $\text{LISEnding}(k)$ we want to focus only on indices $i$ such that $A[i] < A[k]$
- We need to store $\text{LISEnding}(i)$ with each value $A[i]$ stored in the data structure
Augmented Balanced Binary Search Tree

Assume for simplicity that $a_1, a_2, \ldots, a_n$ are distinct numbers.

- We maintain a dynamic balanced binary search tree $T$ which has only $a_1, \ldots, a_{k-1}$ when $\text{LISEnding}(k)$ is getting considered.
Augmented Balanced Binary Search Tree

Assume for simplicity that $a_1, a_2, \ldots, a_n$ are distinct numbers.

- We maintain a dynamic balanced binary search tree $T$ which has only $a_1, \ldots, a_{k-1}$ when $\text{LISEnding}(k)$ is getting considered.
- We can search for $a_k$ in $T$ to obtain a set of subtrees such that each subtree has only numbers smaller than $a_k$. Precisely what we want, and takes $O(\log n)$ time.
Augmented Balanced Binary Search Tree

Assume for simplicity that $a_1, a_2, \ldots, a_n$ are distinct numbers.

- We maintain a dynamic balanced binary search tree $T$ which has only $a_1, \ldots, a_{k-1}$ when $\text{LISEnding}(k)$ is getting considered.
- We can search for $a_k$ in $T$ to obtain a set of subtrees such that each subtree has only numbers smaller than $a_k$. Precisely what we want, and takes $O(\log n)$ time.
- We store with the root of each subtree of $T$ the max $\text{LISEnding}$ value for all indices represented in that subtree.
Augmented Balanced Binary Search Tree

Assume for simplicity that \( a_1, a_2, \ldots, a_n \) are distinct numbers.

- We maintain a dynamic balanced binary search tree \( T \) which has only \( a_1, \ldots, a_{k-1} \) when \( \text{LISEnding}(k) \) is getting considered.
- We can search for \( a_k \) in \( T \) to obtain a set of subtrees such that each subtree has only numbers smaller than \( a_k \). Precisely what we want, and takes \( O(\log n) \) time.
- We store with the root of each subtree of \( T \) the max \( \text{LISEnding} \) value for all indices represented in that subtree.
- Updating tree after computing \( \text{LISEnding}(i) \) requires inserting \( a_i \) into the tree \( T \) and also updating the \( \text{LISEnding} \) values. Can be done in \( O(\log n) \) time. Thus, overall \( O(n \log n) \) time.
A better algorithm

Using only two arrays. Elegant, fast. See Wikipedia article https://en.wikipedia.org/wiki/Longest_increasing_subsequence

Not a first-cut solution.