Dynamic Programming

Lecture 11
October 1, 2015
Dynamic Programming

Dynamic Programming is smart recursion plus memoization
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time *not counting* the time for its recursive calls.

What is an upper bound on the running time of *memoized* version of $\text{foo}(x)$ if $|x| = n$?
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time not counting the time for its recursive calls.

What is an upper bound on the running time of the memoized version of $\text{foo}(x)$ if $|x| = n$? $O(A(n)B(n))$. 
Part I

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
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 Approach: Take 1

**LIS:** Longest increasing subsequence

Can we find a recursive algorithm for **LIS**?

**LIS**($A[1..n]$):
Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

\[ \text{LIS}(A[1..n]): \]

1. **Case 1:** Does not contain \( A[n] \) in which case
   \[ \text{LIS}(A[1..n]) = \text{LIS}(A[1..(n - 1)]) \]

2. **Case 2:** contains \( A[n] \) in which case \( \text{LIS}(A[1..n]) \) is not so clear.

**Observation**

For second case we want to find a subsequence in \( A[1..(n - 1)] \) that is restricted to numbers less than \( A[n] \). This suggests that a more general problem is \( \text{LIS\_smaller}(A[1..n], x) \) which gives the longest increasing subsequence in \( A \) where each number in the sequence is less than \( x \).
Recursive Approach

**LIS(A[1..n])**: the length of longest increasing subsequence in \( A \)

**LIS\_smaller(A[1..n], x)**: length of longest increasing subsequence in \( A[1..n] \) with all numbers in subsequence less than \( x \)

\[
\text{LIS\_smaller}(A[1..n], x) :
\]
\[
\begin{align*}
  &\text{if } (n = 0) \text{ then return } 0 \\
  &m = \text{LIS\_smaller}(A[1..(n - 1)], x) \\
  &\text{if } (A[n] < x) \text{ then} \\
  &\quad m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n - 1)], A[n]))
\end{align*}
\]

Output \( m \)

\[
\text{LIS}(A[1..n]) :
\]
\[
\text{return LIS\_smaller}(A[1..n], \infty)
\]
Example

Sequence: $A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9$
Recursive Approach

\[
\text{LIS}\_\text{smaller}(A[1..n], x) : \\
\quad \text{if } (n = 0) \text{ then return } 0 \\
\quad m = \text{LIS}\_\text{smaller}(A[1..(n - 1)], x) \\
\quad \text{if } (A[n] < x) \text{ then} \\
\qquad m = \max(m, 1 + \text{LIS}\_\text{smaller}(A[1..(n - 1)], A[n])) \\
\text{Output } m
\]

\[
\text{LIS}(A[1..n]) : \\
\quad \text{return } \text{LIS}\_\text{smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS}\_\text{smaller}(A[1..n], \infty) generate?
Recursive Approach

**LIS\_smaller**\((A[1..n], x)\):

1. if \((n = 0)\) then return 0
2. \(m = LIS\_smaller(A[1..(n - 1)], x)\)
3. if \((A[n] < x)\) then
   - \(m = \max(m, 1 + LIS\_smaller(A[1..(n - 1)], A[n]))\)
4. Output \(m\)

\[\text{LIS}(A[1..n]):\]
\[\text{return } LIS\_smaller(A[1..n], \infty)\]

- How many distinct sub-problems will **LIS\_smaller**\((A[1..n], \infty)\) generate? \(O(n^2)\)
Recursive Approach

Recursive Algorithm:

\[
\text{LIS}\_\text{smaller}(A[1..n], x) : \\
\text{if } (n = 0) \text{ then return 0} \\
m = \text{LIS}\_\text{smaller}(A[1..(n - 1)], x) \\
\text{if } (A[n] < x) \text{ then} \\
\quad m = \max(m, 1 + \text{LIS}\_\text{smaller}(A[1..(n - 1)], A[n])) \\
\text{Output } m
\]

\[
\text{LIS}(A[1..n]) : \\
\text{return } \text{LIS}\_\text{smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS}\_\text{smaller}(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion?
Recursive Approach

```python
LIS_smaller(A[1..n], x):
    if (n = 0) then return 0
    m = LIS_smaller(A[1..(n - 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS_smaller(A[1..(n - 1)], A[n]))
    Output m
```

```python
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

- How many distinct sub-problems will $\text{LIS\_smaller}(A[1..n], \infty)$ generate? $O(n^2)$
- What is the running time if we memoize recursion? $O(n^2)$ since each call takes $O(1)$ time to assemble the answers from recursive calls and no other computation.
Recursive Approach

\[
\text{LIS\_smaller}(A[1..n], x): \\
\text{if } (n = 0) \text{ then return 0} \\
m = \text{LIS\_smaller}(A[1..(n-1)], x) \\
\text{if } (A[n] < x) \text{ then} \\
\quad m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n-1)], A[n])) \\
\text{Output } m
\]

\[
\text{LIS}(A[1..n]): \\
\text{return } \text{LIS\_smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate? \(O(n^2)\)

- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.

- How much space for memoization?
Recursive Approach

\[
\text{LIS\_smaller}(A[1..n], x) : \\
\quad \text{if } (n = 0) \text{ then return } 0 \\
\quad m = \text{LIS\_smaller}(A[1..(n - 1)], x) \\
\quad \text{if } (A[n] < x) \text{ then} \\
\quad \quad m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n - 1)], A[n])) \\
\quad \text{Output } m
\]

\[
\text{LIS}(A[1..n]) : \\
\quad \text{return } \text{LIS\_smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
Recursive Algorithm: Take 2

Definition

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

Question: can we obtain a recursive expression?

\[
\begin{align*}
6, 3, 5, 2, 7, 8, 1, 9 \\
\text{LISE}(A[1..8]) &= 5 \\
\text{LISE}(A[1..7]) &= 1 \\
\text{LISE}(A[1..6]) &= 4 \quad \text{(3, 5, 7, 8, 9)} \\
\end{align*}
\]
**Definition**

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

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

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

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

```plaintext
LIS_ending_alg(A[1..n]) :
  if (n = 0) return 0
  m = 1
  for i = 1 to n - 1 do
    if (A[i] < A[n]) then
      m = max(m, 1 + LIS_ending_alg(A[1..i]))
  return m
```

```plaintext
LIS(A[1..n]) :
  return \( \max_{i=1}^{n} \text{LIS}_\text{ending}_\text{alg}(A[1 \ldots i]) \)
```
Recursive Algorithm: Take 2

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

```
LIS(A[1..n]):
  return max_{i=1}^{n} LIS_ending_alg(A[1..i])
```

How many distinct sub-problems will \texttt{LIS_ending_alg(A[1..n])} generate?
Recursive Algorithm: Take 2

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

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

- How many distinct sub-problems will \text{LIS\_ending\_alg}(A[1..n]) generate? \(O(n)\)
Recursive Algorithm: Take 2

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

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

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

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

\[
\text{LIS}(A[1..n]) :}
\]
\[
\text{return } \max_{i=1}^{n}\text{LIS\textunderscore ending\textunderscore alg}(A[1...i])
\]

- How many distinct sub-problems will \text{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
Recursive Algorithm: Take 2

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

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

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

\[
\text{LIS}_\text{ending\_alg}(A[1..n]) : \\
\hspace{1em} \text{if } (n = 0) \text{ return } 0 \\
\hspace{1em} m = 1 \\
\hspace{1em} \text{for } i = 1 \text{ to } n - 1 \text{ do} \\
\hspace{2em} \text{if } (A[i] < A[n]) \text{ then} \\
\hspace{3em} m = \max(m, 1 + \text{LIS}_\text{ending\_alg}(A[1..i])) \\
\hspace{1em} \text{return } m
\]

\[
\text{LIS}(A[1..n]) : \\
\hspace{1em} \text{return } \max_{i=1}^{n} \text{LIS}_\text{ending\_alg}(A[1...i])
\]

- How many distinct sub-problems will \text{LIS}_\text{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
- 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?
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.
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.
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.

Caveat: Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct
Iterative Algorithm via Memoization

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

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

\[
\text{LIS}(A[1..n]) : \\
L = \text{LIS\_ending\_alg}(A[1..n]) \\
\text{return } \text{the maximum value in } L
\]
Simplifying:

\[
\text{LIS}(A[1..n]): \quad \text{Array } L[1..n] \quad (\text{\textit{L[i] stores the value LISEnd}ing}(A[1..i]) \text{ \textit{\asterisk}})
\]

\[
m = 0
\]

\[
\text{for } i = 1 \text{ to } n \text{ do}
\]

\[
L[i] = 1
\]

\[
\text{for } j = 1 \text{ to } i - 1 \text{ do}
\]

\[
\text{if } (A[j] < A[i]) \text{ do}
\]

\[
L[i] = \max(L[i], 1 + L[j])
\]

\[
m = \max(m, L[i])
\]

\[
\text{return } m
\]
Iterative Algorithm via Memoization

Simplifying:

\[
\text{LIS}(A[1..n]):
\begin{align*}
\text{Array } & L[1..n] \ (\ast L[i] \text{ stores the value LISEnd}(A[1..i]) \ast) \\
& m = 0 \\
& \text{for } i = 1 \text{ to } n \text{ do} \\
& \quad L[i] = 1 \\
& \quad \text{for } j = 1 \text{ to } i - 1 \text{ do} \\
& \quad \quad \text{if } (A[j] < A[i]) \text{ do} \\
& \quad \quad \quad L[i] = \max(L[i], 1 + L[j]) \\
& \quad \quad m = \max(m, L[i]) \\
& \text{return } m
\end{align*}
\]

Correctness: Via induction following the recursion

Running time:
Iterative Algorithm via Memoization

Simplifying:

LIS(A[1..n]):
Array L[1..n] (* L[i] stores the value LISEnding(A[1..i]) *)
m = 0
for i = 1 to n do
    L[i] = 1
    for j = 1 to i - 1 do
        if (A[j] < A[i]) do
            L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
return m

Correctness: Via induction following the recursion
Running time: O(n²)
Space:
Iterative Algorithm via Memoization

Simplifying:

\[
\text{LIS}(A[1..n]):
\]

1. Array \( L[1..n] \) (* \( L[i] \) stores the value \( \text{LISEnding}(A[1..i]) \) *)
2. \( m = 0 \)
3. for \( i = 1 \) to \( n \) do
   1. \( L[i] = 1 \)
   2. for \( j = 1 \) to \( i - 1 \) do
         1. \( L[i] = \max(L[i], 1 + L[j]) \)
      2. \( m = \max(m, L[i]) \)
   4. return \( m \)

Correctness: Via induction following the recursion
Running time: \( O(n^2) \)
Space: \( \Theta(n) \)
Iterative Algorithm via Memoization

Simplifying:

LIS\((A[1..n])\):

Array \(L[1..n]\) \((\ast L[i] \text{ stores the value } LISEnding(A[1..i]) \ast)\)

\(m = 0\)

\(\text{for } i = 1 \text{ to } n \text{ do}\)

\(L[i] = 1\)

\(\text{for } j = 1 \text{ to } i - 1 \text{ do}\)

\(\text{if } (A[j] < A[i]) \text{ do}\)

\(L[i] = \max(L[i], 1 + L[j])\)

\(m = \max(m, L[i])\)

\(\text{return } m\)

Correctness: Via induction following the recursion

Running time: \(O(n^2)\)

Space: \(\Theta(n)\)

\(O(n \log n)\) run-time achievable via better data structures.
Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Longest increasing subsequence: 3, 5, 7, 8

$L[i..8] = \text{LISending}(A[i..i])$

$L[1] = 1$

$L[2] = \max (L[1] + 0, 1) = 1$

$L[3] = \max (1, 1 + L[2]) = 1$

$L[4] = \max (1, 1 + 0) = 1$

$L[5] = 1$
Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Longest increasing subsequence: 3, 5, 7, 8

1. \(L[i]\) is value of longest increasing subsequence ending in \(A[i]\)
2. Recursive algorithm computes \(L[i]\) from \(L[1]\) to \(L[i-1]\)
3. Iterative algorithm builds up the values from \(L[1]\) to \(L[n]\)
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual sub-sequence?

Two methods

**Explicit:** For each subproblem find an optimum solution for that subproblem while computing the optimum value for that subproblem. Typically slow but automatic.

**Implicit:** For each subproblem keep track of sufficient information (decision) on how optimum solution for subproblem was computed. Reconstruct optimum solution later via stored information. Typically much more efficient but requires more thought.
Computing Solution: Explicit method for LIS

\textbf{LIS}(A[1..n]):

Array \textbf{L}[1..n] \quad (* \textbf{L}[i] \text{ stores the value } \textbf{LISEnding}(A[1..i]) *)

Array \textbf{S}[1..n] \quad (* \textbf{S}[i] \text{ stores the sequence achieving } \textbf{L}[i] *)

\begin{verbatim}
\begin{algorithm}
m = 0 \hfill h = 0
for i = 1 to n do
    L[i] = 1
    S[i] = [i]
    for j = 1 to i − 1 do
        if (A[j] < A[i]) and (L[i] < 1 + L[j]) do
            L[i] = 1 + L[j]
            S[i] = concat(S[j], [i])
        \end{algorithm}
\end{verbatim}

if (m < L[i]) \quad m = L[i], \quad h = i

return \(m, S[h]\)
Computing Solution: Explicit method for LIS

**LIS**\((A[1..n])\):

Array \(L[1..n]\) (* \(L[i]\) stores the value **LISEnding**\((A[1..i])\) *)
Array \(S[1..n]\) (* \(S[i]\) stores the sequence achieving \(L[i]\) *)

\[m = 0\]
\[h = 0\]

**for** \(i = 1\) **to** \(n\) **do**

\[L[i] = 1\]
\[S[i] = [i]\]

**for** \(j = 1\) **to** \(i - 1\) **do**

**if** \((A[j] < A[i]) \text{ and } (L[i] < 1 + L[j])\) **do**

\[L[i] = 1 + L[j]\]
\[S[i] = \text{concat}(S[j], [i])\]

**if** \((m < L[i])\) \(m = L[i], \ h = i\)

**return** \(m, S[h]\)

Running time: \(O(n^3)\)  Space: \(O(n^2)\). Extra time/space to store, copy...
Computing Solution: Implicit method for LIS

\textbf{LIS}(A[1..n]):

Array \textbf{L}[1..n] (* \textbf{L}[i] stores the value \textbf{LISEnding}(A[1..i]) *)
Array \textbf{D}[1..n] (* \textbf{D}[i] stores how \textbf{L}[i] was computed *)

\begin{verbatim}
m = 0
h = 0
for i = 1 to n do
    L[i] = 1
    D[i] = i
    for j = 1 to i - 1 do
        if (A[j] < A[i]) and (L[i] < 1 + L[j]) do
            L[i] = 1 + L[j]
            D[i] = j
    if (m < L[i]) m = L[i], h = i
\end{verbatim}

\textbf{m} = \textbf{L}[h] is optimum value

Question:
Can we obtain solution from stored \textbf{D} values and \textbf{h}?
**Computing Solution: Implicit method for LIS**

**LIS**(\(A[1..n]\)):

- Array \(L[1..n]\) (* \(L[i]\) stores the value **LIS**\(\text{Ending}(A[1..i])\) *)
- Array \(D[1..n]\) (* \(D[i]\) stores how \(L[i]\) was computed *)

\[
\begin{align*}
m & = 0 \\
h & = 0 \\
\text{for } i & = 1 \text{ to } n \text{ do} \\
& \quad \text{L}[i] = 1 \\
& \quad \text{D}[i] = i \\
& \quad \text{for } j = 1 \text{ to } i - 1 \text{ do} \\
& \quad \quad \text{if } (\text{A}[j] < \text{A}[i]) \text{ and } (\text{L}[i] < 1 + \text{L}[j]) \text{ do} \\
& \quad \quad \quad \text{L}[i] = 1 + \text{L}[j] \\
& \quad \quad \quad \text{D}[i] = j \\
& \quad \quad \text{if } (m < \text{L}[i]) \text{ m = L}[i], \ h = i \\
\end{align*}
\]

\(m = L[h]\) is optimum value

**Question:** Can we obtain solution from stored \(D\) values and \(h\)?
**LIS**(*A[1..n]*):

Array **L[1..n]** (*L[i] stores the value **LISEnding**(*A[1..i]*)*)

Array **D[1..n]** (*D[i] stores how **L[i]** was computed*)

\(m = 0\), \(h = 0\)

for \(i = 1\) to \(n\) do

\[L[i] = 1\]

\[D[i] = 0\]

for \(j = 1\) to \(i - 1\) do

if \((A[j] < A[i])\) and \((L[i] < 1 + L[j])\) do

\[L[i] = 1 + L[j],\]

\[D[i] = j\]

if \((m < L[i])\) \(m = L[i],\)

\(h = i\)

**S** = empty sequence

while \((h > 0)\) do

add \(L[h]\) to front of **S**

\(h = D[h]\)

Output optimum value \(m\), and an optimum subsequence **S**
Computing Solution: Implicit method for LIS

**LIS**(*A[1..n]*):  
Array **L[1..n]** (* L[i] stores the value **LIS**Ending(*A[1..i]*) *)  
Array **D[1..n]** (* D[i] stores how L[i] was computed *)  

\[ m = 0, \ h = 0 \]

**for** i = 1 **to** n **do**

\[ L[i] = 1 \]
\[ D[i] = 0 \]

**for** j = 1 **to** i - 1 **do**

\[ \text{if (A[j] < A[i]) and (L[i] < 1 + L[j]) do} \]
\[ L[i] = 1 + L[j], \ D[i] = j \]

\[ \text{if (m < L[i]) m = L[i], h = i} \]

**S** = empty sequence

**while** (h > 0) **do**

\[ \text{add L[h] to front of S} \]
\[ h = D[h] \]

Output optimum value m, and an optimum subsequence S

**Running time:** \( O(n^2) \)  
**Space:** \( O(n) \).
Dynamic Programming

1. Find a “smart” recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.

2. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. This gives an upper bound on the total running time if we use automatic memoization.

3. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation. This leads to an explicit algorithm.

4. Optimize the resulting algorithm further
Part II

Checking if string in $L^*$
Problem

Input  A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStringinL}(\text{string } x)$ that decides whether $x$ is in $L$

Goal  Decide if $w \in L^*$ using $\text{IsStringinL}(\text{string } x)$ as a black box sub-routine

Example

Suppose $L$ is English and we have a procedure to check whether a string/word is in the English dictionary.

- Is the string “isthisanenglishsentence” in English*?
- Is “stampstamp” in English*?
- Is “zibzzzad” in English*?
Recursive Solution

When is $w \in L^*$?
Recursive Solution

When is $w \in L^*$?

$w \in L^*$ if $w \in L$ or if $w = uv$ where $u \in L$ and $v \in L^*$
Recursive Solution

When is \( w \in L^* \)?

\[ w \in L^* \] if \( w \in L \) or if \( w = uv \) where \( u \in L \) and \( v \in L^* \)

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]) : \\
\text{If (IsStringinL}(A[1..n])) \\
\quad \text{Output YES} \\
\text{Else} \\
\quad \text{For (} i = 1 \text{ to } n - 1 \text{) do} \\
\quad \quad \text{If (IsStringinL}(A[1..i]) \text{ and IsStringinLstar}(A[i + 1..n])) \\
\quad \quad \quad \text{Output YES} \\
\quad \text{Output NO}
\]
Recursive Solution

Assume $w$ is stored in array $A[1..n]$

\[
\text{IsStringinLstar}(A[1..n]):\n\text{If (IsStringinL}(A[1..n]))\n\text{Output YES}\n\text{Else}\n\text{For (i = 1 to n – 1) do}\n\text{If (IsStringinL}(A[1..i]) \text{ and IsStringinLstar}(A[i + 1..n]))\n\text{Output YES}\n\text{Output NO}
\]

- How many distinct sub-problems does $\text{IsStringinLstar}(A[1..n])$ generate?
Recursive Solution

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]): \\
\text{If (IsStringinL}(A[1..n]) \text{))} \\
\quad \text{Output YES} \\
\text{Else} \\
\quad \text{For (} i = 1 \text{ to } n - 1 \text{) do} \\
\qquad \text{If (IsStringinL}(A[1..i]) \text{ and IsStringinLstar}(A[i + 1..n])) \text{)} \\
\qquad \quad \text{Output YES} \\
\quad \text{Output NO}
\]

- How many distinct sub-problems does \text{IsStringinLstar}(A[1..n]) generate? \( O(n) \)
Recursive Solution

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]):
\]

\[
\begin{align*}
\text{If (IsStringinL}(A[1..n])\text{))} \\
\quad \text{Output YES} \\
\text{Else} \\
\quad \text{For (}i = 1 \text{ to } n - 1 \text{ do} \\
\quad \quad \text{If (IsStringinL}(A[1..i]) \text{ and IsStringinLstar}(A[i + 1..n])\text{)} \\
\quad \quad \quad \text{Output YES} \\
\quad \text{Output NO}
\end{align*}
\]

- How many distinct sub-problems does \( \text{IsStringinLstar}(A[1..n]) \) generate? \( O(n) \)
- What is running time of memoized version of \( \text{IsStringinLstar}(A[1..n]) \)?
Recursive Solution

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]) :
\]
\[
\text{If (IsStringinL}(A[1..n])\text{))}
\]
\[
\quad \text{Output YES}
\]
\[
\text{Else}
\]
\[
\quad \text{For (} i = 1 \text{ to } n - 1 \text{) do}
\]
\[
\quad \text{If (IsStringinL}(A[1..i]) \text{ and IsStringinLstar}(A[i + 1..n])\text{))}
\]
\[
\quad \text{Output YES}
\]
\[
\text{Output NO}
\]

- How many distinct sub-problems does \text{IsStringinLstar}(A[1..n])\ generate? \( O(n) \)
- What is running time of memoized version of \text{IsStringinLstar}(A[1..n])? \( O(n^2) \)
Recursive Solution

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]):
\]
\[
\text{If (IsStringinL}(A[1..n]))
\]
\[
\text{Output YES}
\]
\[
\text{Else}
\]
\[
\text{For (i = 1 to n – 1) do}
\]
\[
\text{If (IsStringinL}(A[1..i]) \text{ and IsStringinLstar}(A[i + 1..n]))
\]
\[
\text{Output YES}
\]
\[
\text{Output NO}
\]

- How many distinct sub-problems does \( \text{IsStringinLstar}(A[1..n]) \) generate? \( O(n) \)
- What is running time of memoized version of \( \text{IsStringinLstar}(A[1..n]) \)? \( O(n^2) \)
- What is space requirement of memoized version of \( \text{IsStringinLstar}(A[1..n]) \)?
Recursive Solution

Assume $w$ is stored in array $A[1..n]$

```plaintext
IsStringinLstar(A[1..n]):
    If (IsStringinL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n - 1) do
            If (IsStringinL(A[1..i]) and IsStringinLstar(A[i + 1..n]))
                Output YES
        Output NO
```

- How many distinct sub-problems does $\text{IsStringinLstar}(A[1..n])$ generate? $O(n)$
- What is running time of memoized version of $\text{IsStringinLstar}(A[1..n])$? $O(n^2)$
- What is space requirement of memoized version of $\text{IsStringinLstar}(A[1..n])$? $O(n)$
A variation

Input  A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStringinL}(\text{string } x)$ that decides whether $x$ is in $L$, and non-negative integer $k$

Goal  Decide if $w \in L^k$ using $\text{IsStringinL}(\text{string } x)$ as a black box sub-routine

Example

Suppose $L$ is English and we have a procedure to check whether a string/word is in the English dictionary.

- Is the string “isthisenganlishsentence” in English$^5$?
- Is the string “isthisenganlishsentence” in English$^4$?
- Is “asinineat” in English$^2$?
- Is “asinineat” in English$^4$?
- Is “zibzzzad” in English$^1$?
Recursive Solution

When is $w \in L^k$?
Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L$ and $v \in L^{k-1}$
Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L$ and $v \in L^{k-1}$

Assume $w$ is stored in array $A[1..n]$

\begin{verbatim}
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Output NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n - 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k - 1))
                Output YES
        Output NO
\end{verbatim}
Analysis

IsStringinLk(A[1..n], k):
   If (k = 0)
      If (n = 0) Output YES
      Else Output NO
   If (k = 1)
      Output IsStringinL(A[1..n])
   Else
      For (i = 1 to n − 1) do
         If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
            Output YES
      Output NO

How many distinct sub-problems are generated by IsStringinLk(A[1..n], k)?
Analysis

\[ \text{IsStringinLk}(A[1..n], k): \]
\[ \text{If} \ (k = 0) \]
\[ \quad \text{If} \ (n = 0) \ \text{Output YES} \]
\[ \quad \text{Else Output NO} \]
\[ \text{If} \ (k = 1) \]
\[ \quad \text{Output } \text{IsStringinL}(A[1..n]) \]
\[ \text{Else} \]
\[ \quad \text{For} \ (i = 1 \ \text{to} \ n - 1) \ \text{do} \]
\[ \quad \quad \text{If} \ (\text{IsStringinL}(A[1..i]) \ \text{and} \ \text{IsStringinLk}(A[i + 1..n], k - 1)) \]
\[ \quad \quad \quad \text{Output YES} \]
\[ \quad \text{Output NO} \]

- How many distinct sub-problems are generated by \( \text{IsStringinLk}(A[1..n], k) \)? \( O(nk) \)
Analysis

**IsStringinLk(A[1..n], k):**
- If \(k = 0\)
  - If \(n = 0\) Output YES
  - Else Output NO
- If \(k = 1\)
  - Output **IsStringinL(A[1..n])**
- Else
  - For \(i = 1\) to \(n - 1\) do
    - If (**IsStringinL(A[1..i])** and **IsStringinLk(A[i + 1..n], k - 1)**)
      - Output YES
  - Output NO

- How many distinct sub-problems are generated by **IsStringinLk(A[1..n], k)?** \(O(nk)\)

- How much space?

[Chandra & Manoj (UIUC) CS374 Fall 2015 29 / 32]
Analysis

\textbf{IsStringinLk}(A[1..n], k):

\begin{itemize}
  \item If \((k = 0)\)
    \begin{itemize}
      \item If \((n = 0)\) Output YES
      \item Else Output NO
    \end{itemize}
  \item If \((k = 1)\)
    Output \textbf{IsStringinL}(A[1..n])
  \item Else
    For \((i = 1\) to \(n - 1\)) do
      \begin{itemize}
        \item If \((\textbf{IsStringinL}(A[1..i]) \text{ and } \textbf{IsStringinLk}(A[i + 1..n], k - 1))\)
          Output YES
      \end{itemize}
    Output NO
\end{itemize}

- How many distinct sub-problems are generated by \textbf{IsStringinLk}(A[1..n], k)? \(O(nk)\)
- How much space? \(O(nk)\) pause
- Running time?
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Output NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n − 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
                Output YES
        Output NO

How many distinct sub-problems are generated by IsStringinLk(A[1..n], k)? O(nk)
How much space? O(nk) pause
Running time? O(n^2k)
Another variant

**Question:** What if we want to check if $w \in L_i$ for some $0 \leq i \leq k$?
That is, is $w \in \bigcup_{i=0}^{k} L_i$?
Exercise

Definition

A string is a palindrome if $w = w^R$.

Examples: I, RACECAR, MALAYALAM, DOOFFOOD
Exercise

Definition

A string is a palindrome if $w = w^R$.
Examples: I, RACECAR, MALAYALAM, DOOFFOOD

Problem: Given a string $w$ find the longest subsequence of $w$ that is a palindrome.

Example

MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM has MHYMRORMYHM as a palindromic subsequence
Exercise

Assume $w$ is stored in an array $A[1..n]$

$LPS(A[1..n])$: length of longest palindromic subsequence of $A$.

Recursive expression/code?