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 $\mathbf{foo(x)}$ that takes an input $\mathbf{x}$.

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

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

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

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

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

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

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 $1 \le i_1 < i_2 < \ldots < i_k \le n$.

Definition

A sequence is **increasing** if $a_1 < a_2 < \ldots < a_n$. It is **non-decreasing** if $a_1 \le a_2 \le \ldots \le a_n$. Similarly **decreasing** and **non-increasing**.

Sequences

Example...

Example

1. Sequence: $\mathbf{6, 3, 5, 2, 7, 8, 1, 9}$
2. Subsequence of above sequence: $\mathbf{5, 2, 1}$
3. Increasing sequence: $\mathbf{3, 5, 9, 17, 54}$
4. Decreasing sequence: $\mathbf{34, 21, 7, 5, 1}$
5. Increasing <u>subsequence</u> of the first sequence: $\mathbf{2, 7, 9}$.

Longest Increasing Subsequence Problem

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

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

Longest Increasing Subsequence Problem

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

Goal Find an **increasing subsequence** $\mathbf{a_{i_1}, a_{i_2}, \ldots, a_{i_k}}$ 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?

$\mathrm{LIS}(\mathbf{A[1..n]})$:

1. Case 1: Does not contain $\mathbf{A[n]}$ in which case $\mathrm{LIS}(\mathbf{A[1..n]}) = \mathrm{LIS}(\mathbf{A[1..(n-1)]})$
2. Case 2: contains $\mathbf{A[n]}$ in which case $\mathrm{LIS}(\mathbf{A[1..n]})$ is not so clear.

Observation

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

Recursive Approach

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

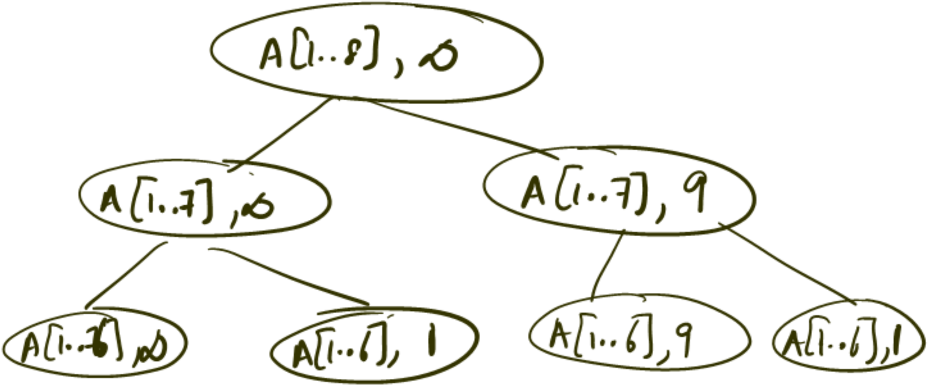
LIS_smaller(A[1..n], x): length of longest increasing subsequence in **A[1..n]** with all numbers in subsequence less than **x**

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

```
LIS(A[1..n]) :
    return LIS_smaller(A[1..n], $\infty$)
```

Example

Sequence: $\mathbf{A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9}$



Recursive Approach

```
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
```

```
LIS(A[1..n]) :
    return LIS_smaller(A[1..n], $\infty$)
```

- How many distinct sub-problems will **LIS_smaller($A[1..n], \infty$)** generate?

Recursive Approach

```
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
```

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

- How many distinct sub-problems will **LIS_smaller(A[1..n], ∞)** generate? $\mathbf{O(n^2)}$

Recursive Approach

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

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

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

Recursive Approach

```
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
```

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

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

Recursive Approach

```
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
```

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

- How many distinct sub-problems will **LIS_smaller(A[1..n], ∞)** generate? $\mathbf{O(n^2)}$
- What is the running time if we memoize recursion? $\mathbf{O(n^2)}$ since each call takes $\mathbf{O(1)}$ time to assemble the answers from to recursive calls and no other computation.
- How much space for memoization?

Recursive Approach

```
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
```

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

- How many distinct sub-problems will **LIS_smaller(A[1..n], ∞)** generate? $\mathbf{O(n^2)}$
- What is the running time if we memoize recursion? $\mathbf{O(n^2)}$ since each call takes $\mathbf{O(1)}$ time to assemble the answers from to recursive calls and no other computation.
- How much space for memoization? $\mathbf{O(n^2)}$

Recursive Algorithm: Take 2

Definition

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

Question: can we obtain a recursive expression?

6, 3, 5, 2, 7, 8, 1, 9

LISE(A[1..8]) = 5 (3, 5, 7, 8, 9)

LISE(A[1..7]) = 1 (1)

(A[1..6]) = 4 (3, 5, 7, 8)

Recursive Algorithm: Take 2

Definition

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

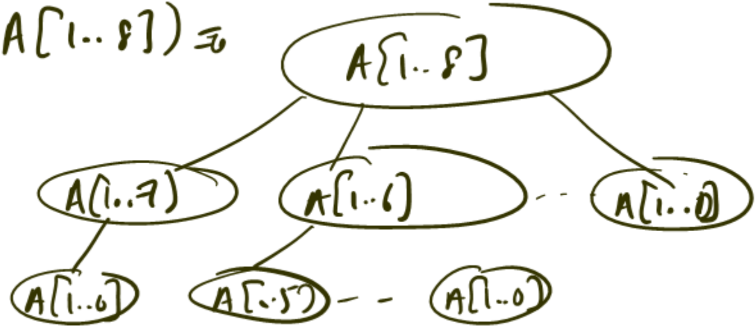
Question: can we obtain a recursive expression?

$$\mathbf{LISEnding(A[1..n])} = \max_{\mathbf{i:A[i]<A[n]}} \Big(\mathbf{1 + LISEnding(A[1..i])}\Big)$$

Example

Sequence: $\mathbf{A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9}$

List (A[1..8]) =



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
            $m = \max\left(m,\ 1 + \text{LIS\_ending\_alg}(A[1..i])\right)$
    return m
```

```
LIS(A[1..n]):
        return $\max_{i=1}^{n}$ LIS_ending_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 **LIS_ending_alg(A[1..n])** generate?

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 **LIS_ending_alg(A[1..n])** generate? $O(n)$

Recursive Algorithm: Take 2

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

```
$\text{LIS}(A[1..n])$:
        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? $O(n)$
- What is the running time if we memoize recursion?

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
            $m = \max\left(m,\ 1 + LIS\_ending\_alg(A[1..i])\right)$
    return $m$
```

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

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

Recursive Algorithm: 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 **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
- How much space for memoization?

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 **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
- 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 **LIS_ending_alg(A[1..i])** iteratively in a bottom up fashion.

```
LIS_ending_alg(A[1..n]):
    Array L[1..n]   (* L[i] = value of LIS_ending_alg(A[1..i]) *)
    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])
    return L
```

```
LIS(A[1..n]):
        L = LIS_ending_alg(A[1..n])
        return the maximum value in L
```

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
```

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:

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: $O(n^2)$

Space:

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: $O(n^2)$

Space: $\Theta(n)$

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: $O(n^2)$

Space: $\Theta(n)$

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

Example

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1, 9
2. Longest increasing subsequence: 3, 5, 7, 8, 9

$L[1..8]$ $L[i] = \text{LISending}(A[1..i])$

$L[1] = 1$

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

$L[3] = \max(1, 1+L[2], 1+0) =$

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

$L[5] =$

Example

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]**

Computing Solutions

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual sub-sequence?
2. Two methods
    1. **Explicit:** For each subproblem find an optimum solution for that subproblem while computing the optimum value for that subproblem. Typically slow but automatic.
    2. **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

```
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]) and (L[i] < 1 + L[j]) do
                L[i] = 1 + L[j]
                S[i] = concat(S[j], [i])

        if (m < L[i]) m = L[i], 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 ($A[j] < A[i]$) and ($L[i] < 1 + L[j]$) do
                $L[i] = 1 + L[j]$
                $S[i] =$ 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

```
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
        $L[i] = 1$
        $D[i] = i$
        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 ($m < L[i]$) $m = L[i]$, $h = i$

    $m = L[h]$ is optimum value
```

Computing Solution: Implicit method for LIS

```
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
        $L[i] = 1$
        $D[i] = i$
        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 ($m < L[i]$) $m = L[i]$, $h = i$

    $m = L[h]$ is optimum value
```

Question: Can we obtain solution from stored **D** values and **h**?

Computing Solution: Implicit method for LIS

```
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 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 $S$
```

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^* L^*

Problem

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

Goal Decide if $\mathbf{w} \in \mathbf{L}^*$ using **IsStringinL(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***?

" $a_1 a_2 - a_n$

Recursive Solution

When is $\mathbf{w} \in \mathbf{L}^*$?

Recursive Solution

When is $\mathbf{w} \in \mathbf{L}^*$?

$\mathbf{w} \in \mathbf{L}^*$ if $\mathbf{w} \in \mathbf{L}$ or if $\mathbf{w} = \mathbf{uv}$ where $\mathbf{u} \in \mathbf{L}$ and $\mathbf{v} \in \mathbf{L}^*$

Recursive Solution

When is $\mathbf{w} \in \mathbf{L}^*$?

$\mathbf{w} \in \mathbf{L}^*$ if $\mathbf{w} \in \mathbf{L}$ or if $\mathbf{w} = \mathbf{uv}$ where $\mathbf{u} \in \mathbf{L}$ and $\mathbf{v} \in \mathbf{L}^*$

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

```
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
```

Recursive Solution

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

```
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 **IsStringinLstar($A[1..n]$)** generate?

Recursive Solution

Assume $\mathbf{w}$ is stored in array $\mathbf{A[1..n]}$

```
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 **IsStringinLstar(A[1..n])** generate? $\mathbf{O(n)}$

Recursive Solution

Assume $\mathbf{w}$ is stored in array $\mathbf{A[1..n]}$

```
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 **IsStringinLstar(A[1..n])** generate? $\mathbf{O(n)}$
- What is running time of memoized version of **IsStringinLstar(A[1..n])**?

Recursive Solution

Assume $\mathbf{w}$ is stored in array $\mathbf{A[1..n]}$

```
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 **IsStringinLstar(A[1..n])** generate? $\mathbf{O(n)}$
- What is running time of memoized version of **IsStringinLstar(A[1..n])**? $\mathbf{O(n^2)}$

Recursive Solution

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

```
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 **IsStringinLstar**$(A[1..n])$ generate? $O(n)$
- What is running time of memoized version of **IsStringinLstar**$(A[1..n])$? $O(n^2)$
- What is space requirement of memoized version of **IsStringinLstar**$(A[1..n])$?

Recursive Solution

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

```
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 **IsStringinLstar(A[1..n])** generate? **$O(n)$**
- What is running time of memoized version of **IsStringinLstar(A[1..n])**? **$O(n^2)$**
- What is space requirement of memoized version of **IsStringinLstar(A[1..n])**? **$O(n)$**

A variation

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

Goal Decide if $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$ using **IsStringinL(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 $\mathbf{English}^5$?
- Is the string "isthisanenglishsentence" in $\mathbf{English}^4$?
- Is "asinineat" in $\mathbf{English}^2$?
- Is "asinineat" in $\mathbf{English}^4$?
- Is "zibzzzad" in $\mathbf{English}^1$?

Recursive Solution

When is $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$?

Recursive Solution

When is $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$?

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

$\mathbf{k} = \mathbf{1}$: $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$ iff $\mathbf{w} \in \mathbf{L}$

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

Recursive Solution

When is $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$?

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

$\mathbf{k} = \mathbf{1}$: $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$ iff $\mathbf{w} \in \mathbf{L}$

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

Assume $\mathbf{w}$ is stored in array $\mathbf{A[1..n]}$

```
IsStringinLk(A[1..n], k):
    If ($k = 0$)
        If ($n = 0$) Output YES
        Else Ouput 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
```

Analysis

```
IsStringinLk(A[1..n], k):
    If ($k = 0$)
        If ($n = 0$) Output YES
        Else Ouput 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

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

    Output NO
```

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

Analysis

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput 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?

Analysis

```
IsStringinLk(A[1..n], k):
    If ($k = 0$)
        If ($n = 0$) Output YES
        Else Ouput 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?

Analysis

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

    Output NO
```

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

Another variant

Question: What if we want to check if $\mathbf{w} \in \mathbf{L}^{\mathbf{i}}$ for some $\mathbf{0} \leq \mathbf{i} \leq \mathbf{k}$? That is, is $\mathbf{w} \in \cup_{\mathbf{i}=\mathbf{0}}^{\mathbf{k}} \mathbf{L}^{\mathbf{i}}$?

Exercise

Definition

A string is a palindrome if $\mathbf{w} = \mathbf{w}^{\mathbf{R}}$.
Examples: **I**, **RACECAR**, **MALAYALAM**, **DOOFFOOD**

Exercise

Definition

A string is a palindrome if $\mathbf{w} = \mathbf{w}^{\mathbf{R}}$.
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?