BBM402-Lecture 4: Dynamic Programming: Longest Increasing Subsequence, String splitting

Lecturer: Lale Özkahya

Resources for the presentation:
https://courses.engr.illinois.edu/cs374/fa2016/lectures.html
https://courses.engr.illinois.edu/cs473/fa2016/lectures.html
https://courses.engr.illinois.edu/cs374/fa2015/lectures.html

Fibonacci

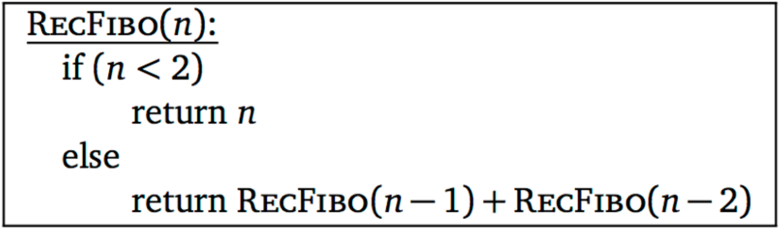
- Fibonacci Numbers (circa 13 th century)

- $F_n =$
$$\begin{cases} 0 \text{ if } n=0 \\ 1 \text{ if } n=1 \\ F_{n-1}+F_{n-2} \text{ o/w} \end{cases}$$

Given n, how long does it take to compute F_n?

Fibonacci

- Translates line by line to code:

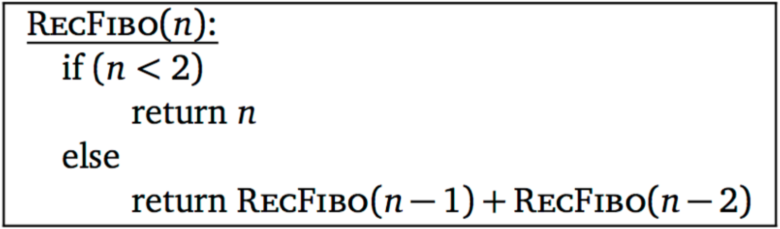


```
RecFibo($n$):
    if ($n < 2$)
        return $n$
    else
        return RecFibo($n-1$) + RecFibo($n-2$)
```

We will move from mathematical function format to recursive program a lot!

Fibonacci

- Translates line by line to code:



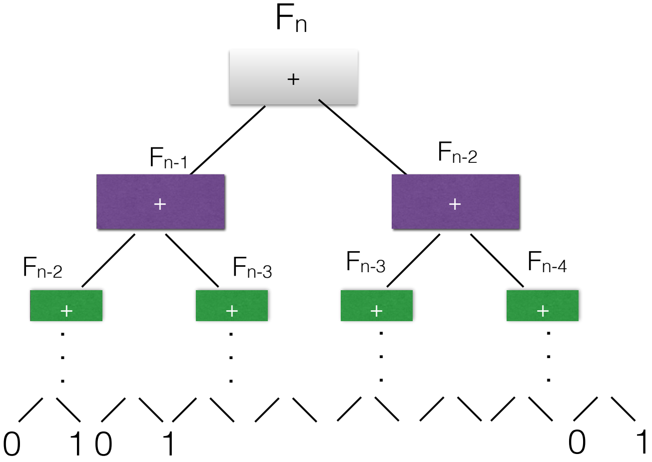
```
RecFibo(n):
    if ($n < 2$)
        return $n$
    else
        return RecFibo($n-1$) + RecFibo($n-2$)
```

Running time? (backtracking recurrence)

$$T(n)=T(n-1)+T(n-2)+O(1)$$

$$=\Theta(F_n) = \Theta(1.618^n) = \Theta(((\sqrt{5}+1)/2)^n)$$

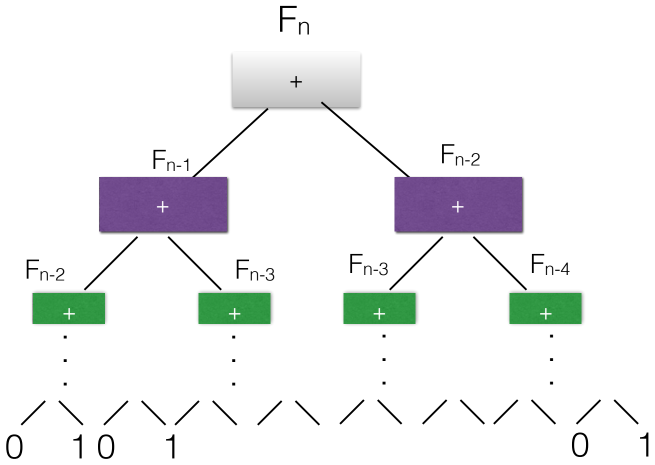
Running time via Rec Tree



Leaves are always 0 or 1.
How many 1's? How many 0s?

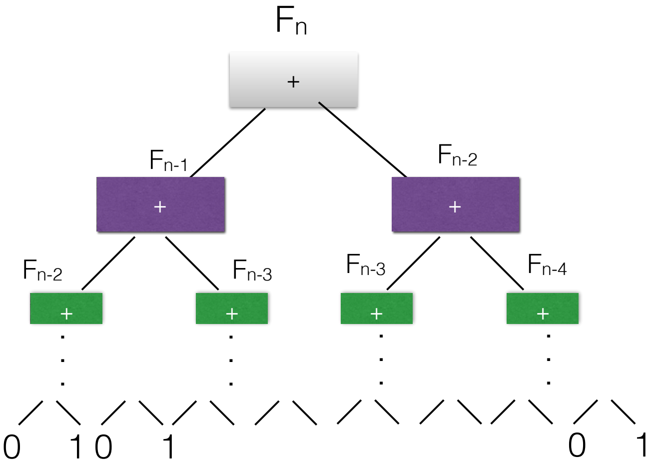
There are F_n 1s and F_{n-1} 0s
F_{n+1} leaves total!

Running time via Rec Tree



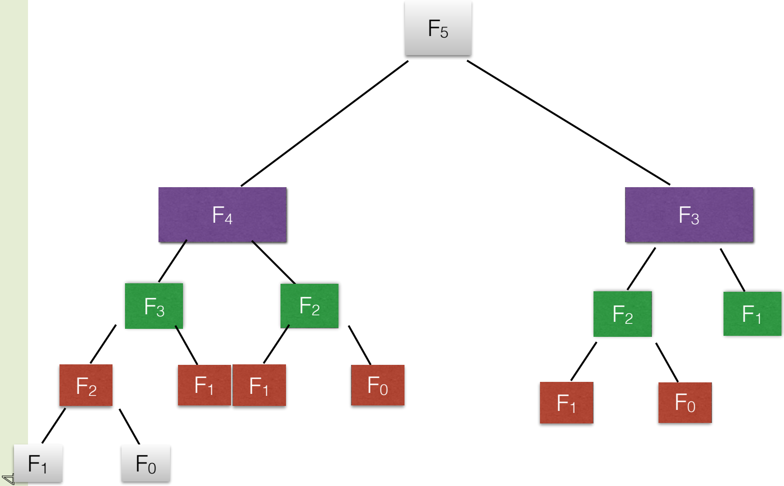
How many intermediate nodes does a full binary tree with m leaves have?

Running time via Rec Tree

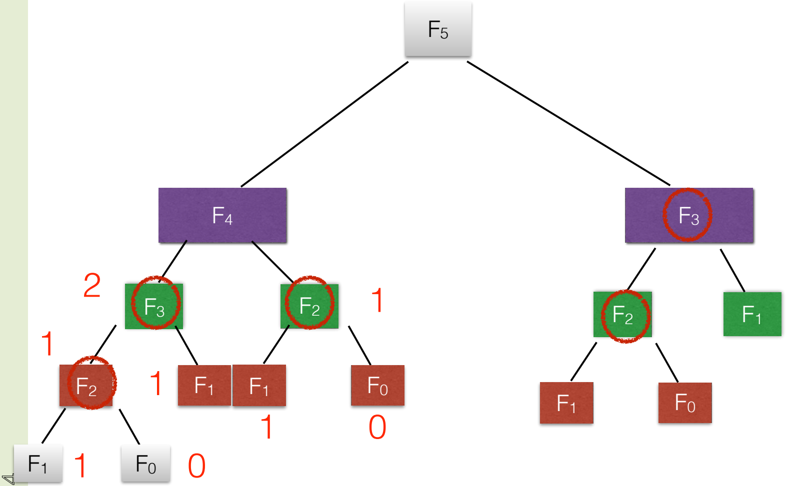


$2F_{n+1} - 1$ nodes (additions)

Running time via Rec Tree

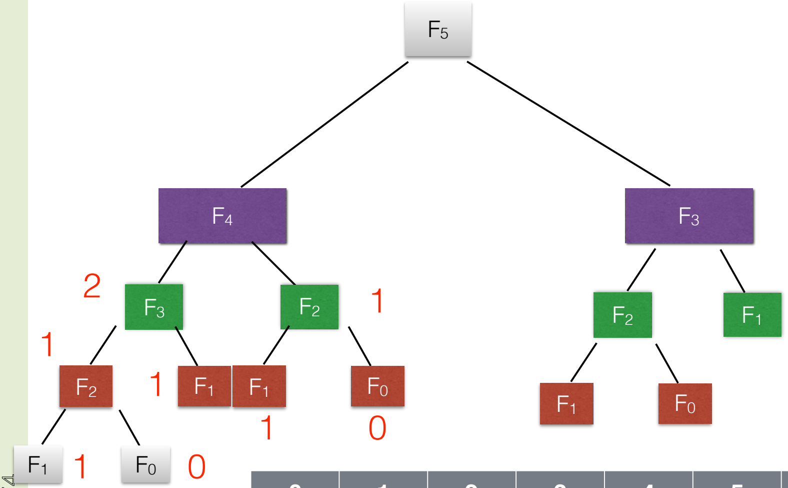


Running time via Rec Tree



Keep an array to remember the previous values!

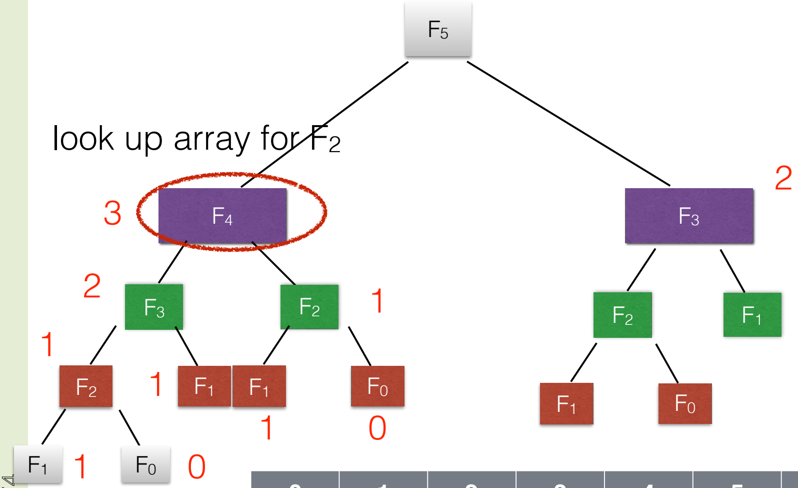
Running time via Rec Tree



F

0	1	2	3	4	5	...
0	1	1	2			

Running time via Rec Tree

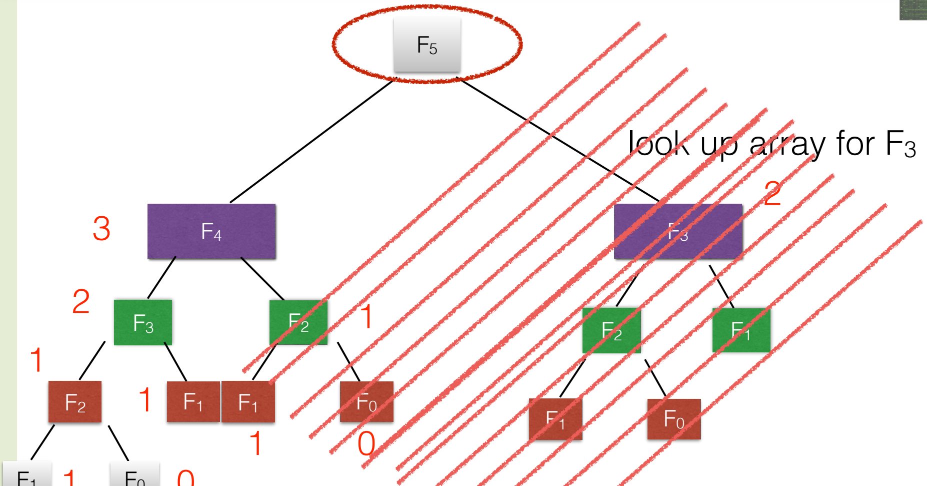


look up array for F_2

F

0	1	2	3	4	5	...
0	1	1	2	3		

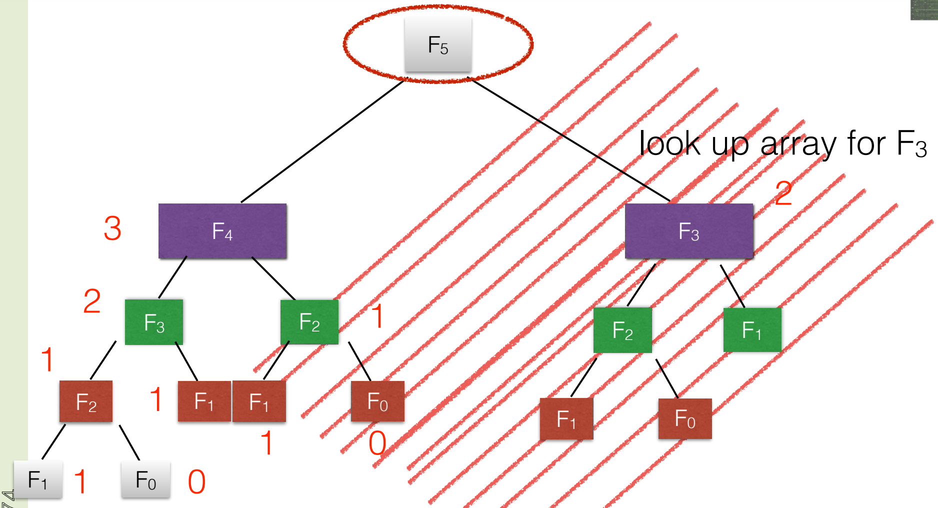
Running time via Rec Tree



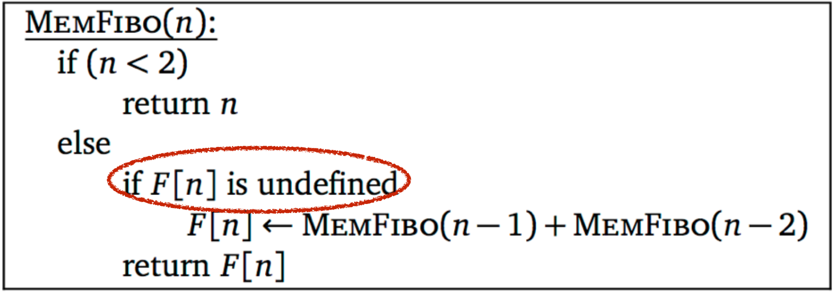
look up array for F3

	0	1	2	3	4	5	...
F	0	1	1	2	3	5	

Running time via Rec Tree

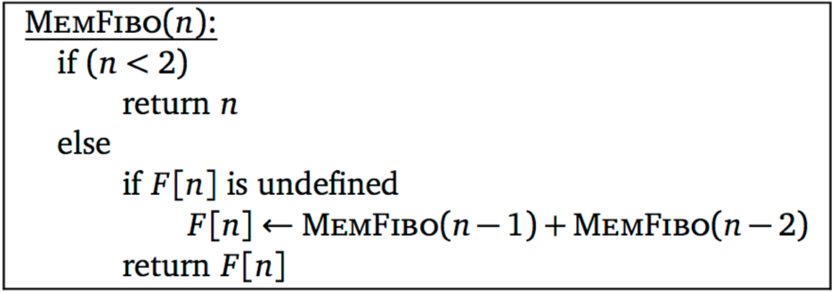


Memoization= when I look at the table to see the values I computed before



```
MEMFIBO($n$):
    if ($n < 2$)
        return $n$
    else
        if $F[n]$ is undefined
            $F[n] \leftarrow$ MEMFIBO($n-1$) + MEMFIBO($n-2$)
        return $F[n]$
```

Given any recursive backtracking algorithm,
you can add memoization and will save time, provided the
subproblems repeat

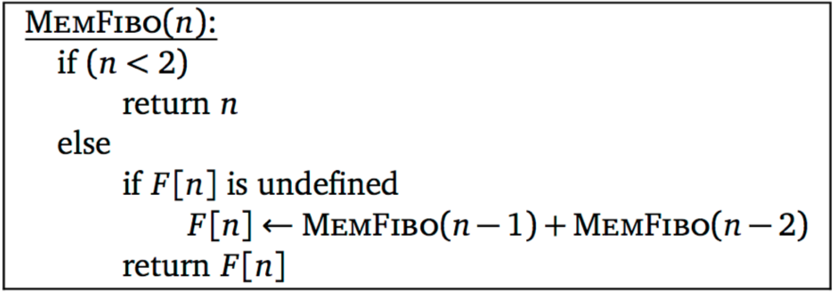


```
MemFibo($n$):
    if ($n < 2$)
        return $n$
    else
        if $F[n]$ is undefined
            $F[n] \leftarrow$ MemFibo($n-1$) + MemFibo($n-2$)
        return $F[n]$
```

How many times did I have to call the recursive function?
exponential!

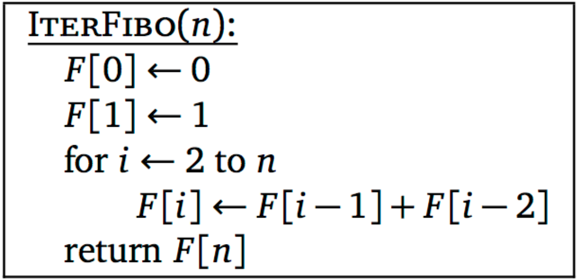
How many different values did I have to compute?
O(n)!

Memoization decreases running time : performs only O(n) additions, exponential improvement



```
MEMFIBO($n$):
  if ($n < 2$)
      return $n$
  else
      if $F[n]$ is undefined
          $F[n] \leftarrow$ MEMFIBO($n-1$) + MEMFIBO($n-2$)
      return $F[n]$
```

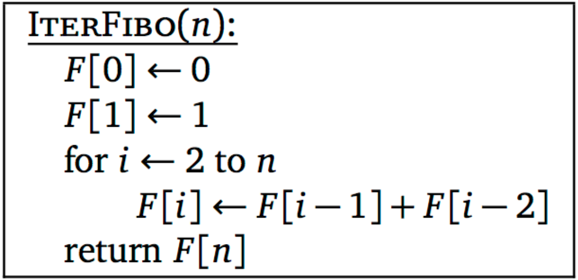
Memoized algorithm fills in the table from left to right.
Why not just do that?



```
IterFibo($n$):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

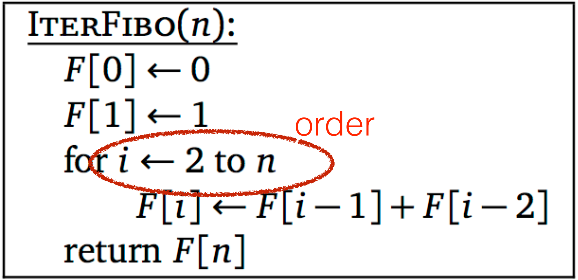
Memoized algorithm fills in the table from left to right.
Why not just do that?

We get an iterative algorithm



```
ITERFIBO($n$):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

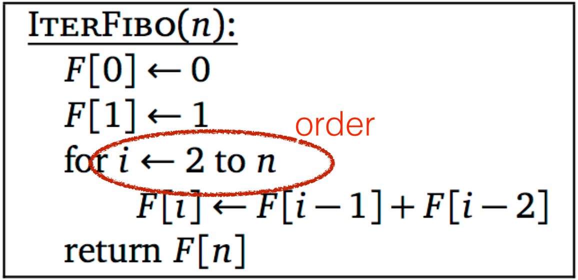
- Clear that the number of additions it does it O(n).
- In practice this is faster than memoized algo, cause we don’t use stack/ look up the table etc.



```
ITERFIBO($n$):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

order

- Structure mirrors the recurrence
- Only subtle thing is that we want to fill in the array in increasing order.



```
ITERFIBO($n$):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

order

- This is Dynamic Programing Algorithm!
- Dynamic Programming= pretend to do Memoization but do it on purpose

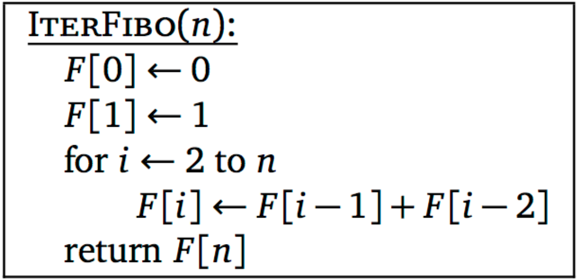
- Memoization: accidentally use something efficient
- Backwards induction =Dynamic Programming

Dynamic Programming

- Dynamic programming is about smart recursion.
- Not about filling out tables!
- How do I solve the problem, how do I not repeat work, then how to fill up my data structure.

Dynamic Programming

- How can I speed up my algorithm?

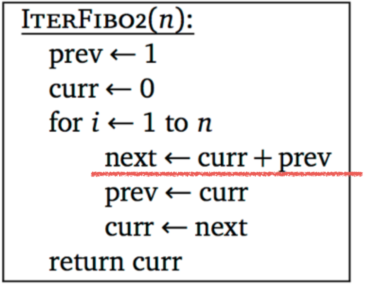


```
ITERFIBO($n$):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

- I only need to keep my last two elements of the array.
- Even more efficient algorithm

Dynamic Programming

- How can I speed up my algorithm?

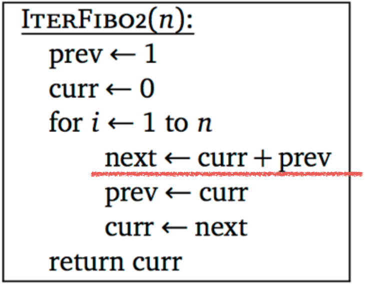


```
ITERFIBO2($n$):
    prev ← 1
    curr ← 0
    for $i$ ← 1 to $n$
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

- I only need to keep my last two elements of the array.
- Even more efficient algorithm
- Where is the recursion?

Dynamic Programming

- How can I speed up my algorithm?

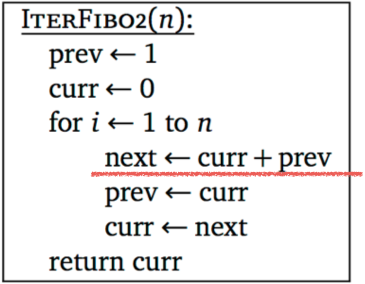


```
ITERFIBO2($n$):
    prev ← 1
    curr ← 0
    for $i$ ← 1 to $n$
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

- I only need to keep my last two elements of the array.
- Even more efficient algorithm
- Where is the recursion?
- Saves space, sometimes important

Dynamic Programming

- How can I speed up my algorithm?

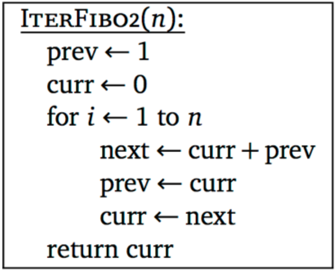


```
ITERFIBO2($n$):
    prev ← 1
    curr ← 0
    for $i$ ← 1 to $n$
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

- Is this the fastest Algorithm for Fibonacci?

Dynamic Programming

- How can I speed up my algorithm?



```
ITERFIBO2(n):
    prev ← 1
    curr ← 0
    for i ← 1 to n
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x + y \end{bmatrix}$$

This matrix vector multiplication does exactly the same thing as one iteration of the loop!

What to do to compute the nth Fibonacci number?

Dynamic Programming

- How can I speed up my algorithm?

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

Compute the nth power of the matrix.

- With repeated squaring, O(logn) multiplications
- Compute F_n in O(logn) arithmetic operations
- Double exponential speedup!

Dynamic Programming

- How can I speed up my algorithm?

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x + y \end{bmatrix}$$

Compute the nth power of the matrix.

- But how many bits is the nth Fibonacci number?
- O(n)!
- Can’t perform arbitrary precision arithmetic in constant time

Longest Increasing Subsequence (LIS)

- 3 1 4 1 5 9 2 6 5 3 8 2 7 9 4 6 1 0 4 8

Longest Increasing Subsequence (LIS)

- 3 1 4 1 5 9 2 6 5 3 8 2 7 9 4 6 1 0 4 8

- LIS(A[1…n],p) = length of LIS of A[1…n] where everything is bigger than p

Longest Increasing Subsequence (LIS)

- 3 1 4 1 5 9 2 6 5 3 8 2 7 9 4 6 1 0 4 8
- $$\mathrm{LIS}(A[1\ldots n],p) = \begin{cases} 0 \text{ if } n=0 \\ \mathrm{LIS}(A[2\ldots n],p) \text{ if } A[1] \le p \\ \mathrm{MAX}\ \{\ \mathrm{LIS}(A[2\ldots n],p) \\ \qquad 1+\mathrm{LIS}(A[2\ldots n],A[1])\} \end{cases}$$

Longest Increasing Subsequence (LIS)

- $$\text{LIS}(A[1\ldots n],p) = \begin{cases} 0 \text{ if } n=0 \\ \text{LIS}(A[2\ldots n],p) \text{ if } A[1] \le p \\ \text{MAX}\,\{\, \text{LIS}(A[2\ldots n],p) \\ \qquad 1+\text{LIS}(A[2\ldots n],A[1])\} \end{cases}$$
- The argument p is always either $-\infty$ or and element of the array A
- Add $A[0]=-\infty$
- We can identify any recursive subproblem with two array indices.
- LIS(i,j) = length or LIS of A[j…n] with all elements larger tha A[i]

Longest Increasing Subsequence (LIS)

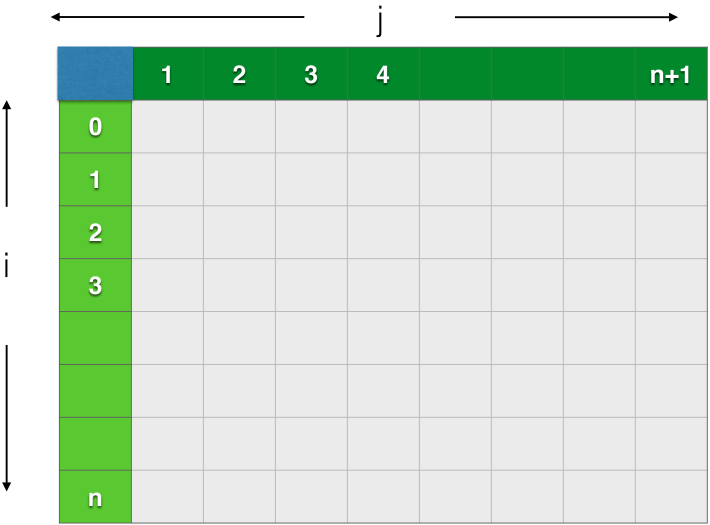
For i<j

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$

- LIS(i,j) = length or LIS of A[j…n] with all elements larger tha A[i]
- We want to compute LIS(0,1)
- Memoize? what data structure to use?
- Two dimensional Array **LIS[0…n,1…n+1]**

For i<j

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$



	1	2	3	4				n+1
0								
1								
2								
3								
n								

For i<j

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$

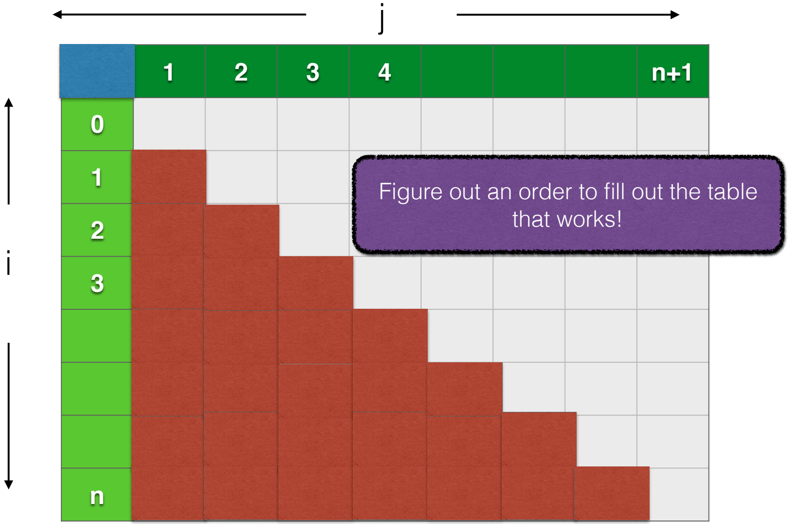
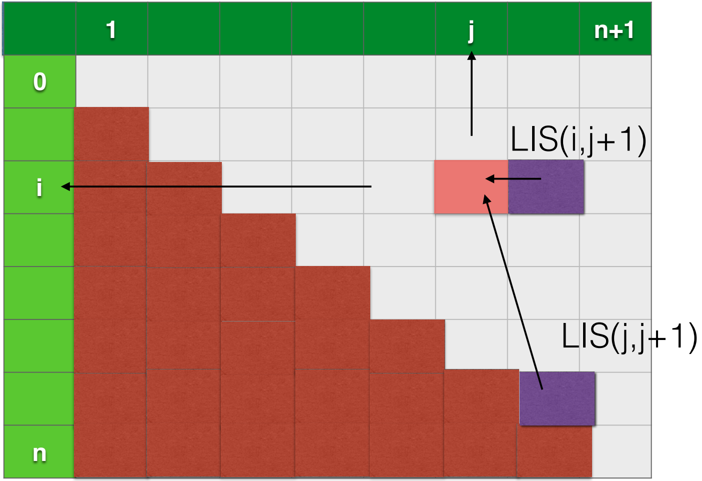


Figure out an order to fill out the table that works!

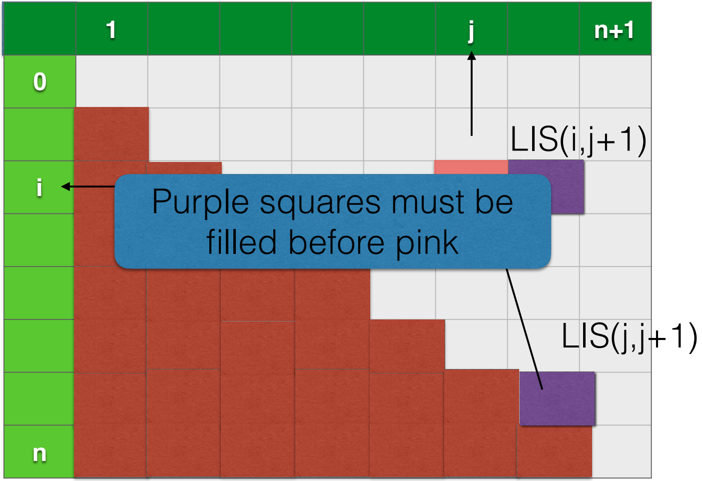
For i<j

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS\boxed{(i,j+1)} & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS\boxed{(j,j+1)}\} & \text{otherwise} \end{cases}$$



For i<j

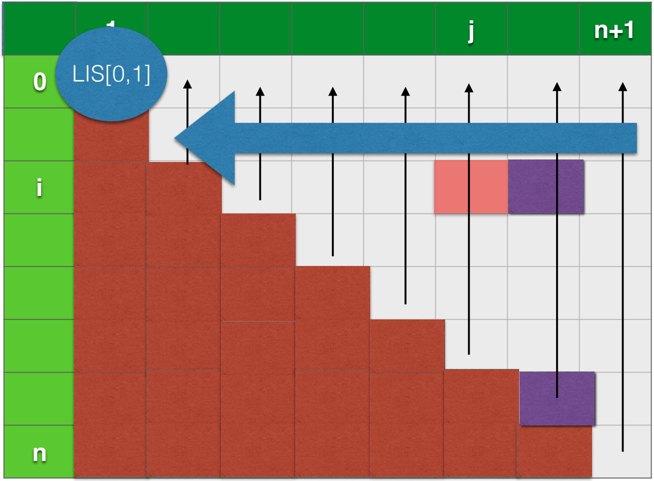
$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$



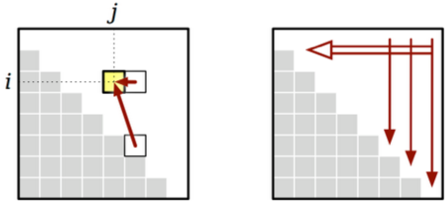
Purple squares must be filled before pink

For $i<j$

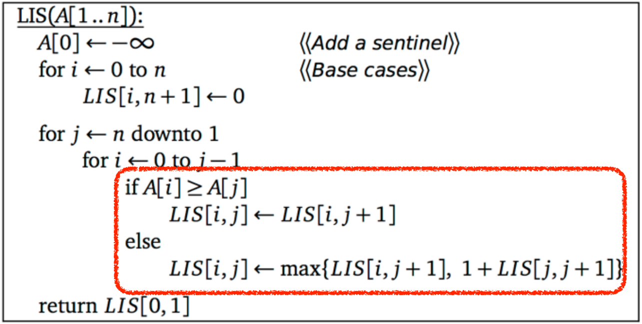
$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS\boxed{(i, j+1)} & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS\boxed{(j,j+1)}\} & \text{otherwise} \end{cases}$$



Longest Increasing Subsequence (LIS)



doesn't matter what order I fill the columns in

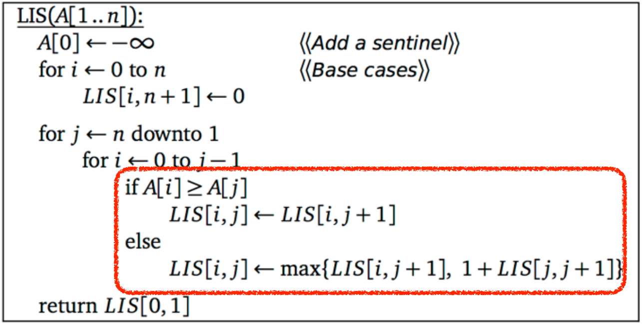


```
LIS(A[1..n]):
    $A[0] \leftarrow -\infty$                    ⟨⟨Add a sentinel⟩⟩
    for $i \leftarrow 0$ to $n$                  ⟨⟨Base cases⟩⟩
        $LIS[i, n+1] \leftarrow 0$

    for $j \leftarrow n$ downto 1
        for $i \leftarrow 0$ to $j-1$
            if $A[i] \geq A[j]$
                $LIS[i, j] \leftarrow LIS[i, j+1]$
            else
                $LIS[i, j] \leftarrow \max\{LIS[i, j+1],\ 1 + LIS[j, j+1]\}$
    return $LIS[0, 1]$
```

Longest Increasing Subsequence (LIS)

- Running time?
- O(n^2)
- Two nested for loops
- How man values are there in the recurrence?



```
LIS(A[1..n]):
    $A[0] \leftarrow -\infty$                  ⟨⟨Add a sentinel⟩⟩
    for $i \leftarrow 0$ to $n$               ⟨⟨Base cases⟩⟩
        $LIS[i, n+1] \leftarrow 0$
    for $j \leftarrow n$ downto 1
        for $i \leftarrow 0$ to $j-1$
            if $A[i] \geq A[j]$
                $LIS[i, j] \leftarrow LIS[i, j+1]$
            else
                $LIS[i, j] \leftarrow \max\{LIS[i, j+1],\ 1 + LIS[j, j+1]\}$
    return $LIS[0, 1]$
```

Longest Increasing Subsequence (LIS)

For i<j

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$

- As general rule of thumb:
- # variables on the left =space O(n^2) array for i,j taking n values each
- # variables on the right =time O(n^2)

Dynamic Programming

General Recipe for DP

- **Step 1**: Find Backtracking Recursive algorithm (e.g. for LIS we leveraged the recursive def. Either empty or there is something that comes first) (6 pts)

- **Step 2**: Identify the subproblems (e.g. indices i,j for LIS), need english description

- **Step 3**: Analyze time and space

- **Step 4**: Choose a memoization data structure (e.g. two dim array)

- **Step 5**: Find evaluation order (draw picture!!!)

Dynamic Programming

General Recipe for DP

- **Step 3**: Analyze time and space
- **Step 6**: write iterative pseudocode

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 **foo(x)** that takes an input **x**.

- On input of size **n** the number of *distinct* sub-problems that **foo(x)** generates is at most **A(n)**
- **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 **foo(x)** if $|\mathbf{x}| = \mathbf{n}$?

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question: Suppose we have a recursive program **foo(x)** that takes an input **x**.

- On input of size **n** the number of *distinct* sub-problems that **foo(x)** generates is at most **A(n)**
- **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 **foo(x)** if $|\mathbf{x}| = \mathbf{n}$? **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: $\mathbf{6, 3, 5, 2, 7, 8, 1, 9}$
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**

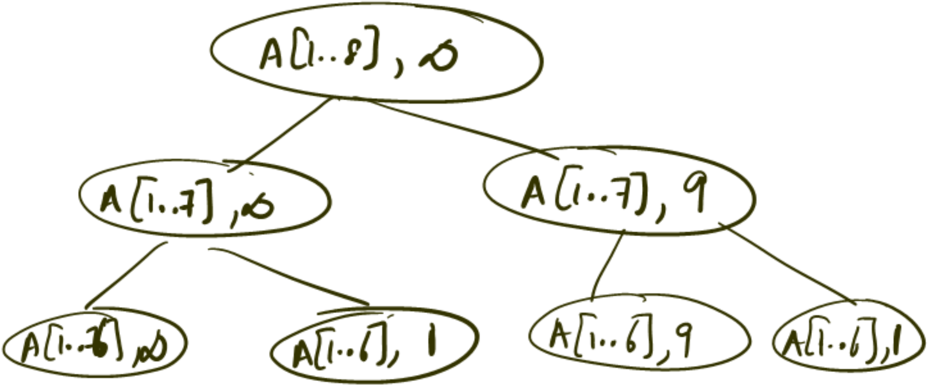
LIS_smaller(A[1..n], x): length of longest increasing subsequence in **A[1..n]** with all numbers in subsequence less than **x**

```
LIS_smaller(A[1..n], x):
    if ($n = 0$) then return 0
    $m$ = 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], ∞)
```

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

Recursive Approach

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

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

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

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? **$O(n^2)$**
- 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? $O(n^2)$
- What is the running time if we memoize recursion? $O(n^2)$ since each call takes $O(1)$ time to assemble the answers from 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], $\infty$)
```

- 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]**.

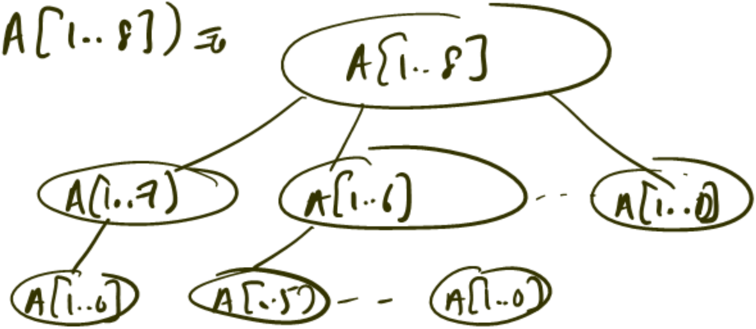
Question: can we obtain a recursive expression?

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

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

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

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

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

- 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

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

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

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

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? $O(n^2)$ 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
            $m = \max\left(m,\ 1 + \text{LIS\_ending\_alg}(A[1..i])\right)$
    return $m$
```

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

- 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
- 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

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

$L[1] = 1$

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

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

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

$L[5] =$

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

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 $\mathbf{w}$ 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
```

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?

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)**

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])**?

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

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

Recursive Solution

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

$\mathbf{k = 0}$: $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$ iff $\mathbf{w} = \epsilon$
$\mathbf{k = 1}$: $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$ iff $\mathbf{w} \in \mathbf{L}$
$\mathbf{k > 1}$: $\mathbf{w} \in \mathbf{L}^{\mathbf{k}}$ if $\mathbf{w} = \mathbf{uv}$ with $\mathbf{u} \in \mathbf{L}$ and $\mathbf{v} \in \mathbf{L}^{\mathbf{k-1}}$

Recursive Solution

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

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

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

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

```
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)**

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 $\mathbf{w} \in \mathbf{L^i}$ for some $\mathbf{0} \leq \mathbf{i} \leq \mathbf{k}$? That is, is $\mathbf{w} \in \cup_{\mathbf{i=0}}^{\mathbf{k}} \mathbf{L^i}$?

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?

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**:

$$\mathbf{\underline{F}OOD} \rightarrow \mathbf{MO\underline{O}D} \rightarrow \mathbf{MON\underline{O}D} \rightarrow \mathbf{MONE\underline{D}} \rightarrow \mathbf{MONEY}$$

Edit Distance: Alternate View

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.

F	**O**	**O**		**D**
M	**O**	**N**	**E**	**Y**

Edit Distance: Alternate View

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.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$.

Edit Distance: Alternate View

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.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

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 $Y = \beta y$

α, β: strings. x and y single characters.

Possible alignments between X and Y

α	x
β	y

or

α	x
βy	

or

αx	
β	y

Observation

Prefixes must have optimal alignment!

Edit Distance

Basic observation

Let $X = \alpha x$ and $Y = \beta y$

α, β: strings. x and y single characters.

Possible alignments between X and Y

α	x
β	y

or

α	x
βy	

or

αx	
β	y

Observation

Prefixes must have optimal alignment!

$$EDIST(X, Y) = \min \begin{cases} EDIST(\alpha, \beta) + [x = y] \\ 1 + EDIST(\alpha, Y) \\ 1 + EDIST(X, \beta) \end{cases}$$

Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$

```
$EDIST(A[1..m], B[1..n])$
    If ($m = 0$) return $n$
    If ($n = 0$) return $m$
    $m_1 = 1 + EDIST(A[1..(m-1)], B[1..n])$
    $m_2 = 1 + EDIST(A[1..m], B[1..(n-1)]))$
    If $(A[m] = B[n])$ then
        $m_3 = EDIST(A[1..(m-1)], B[1..(n-1)])$
    Else
        $m_3 = 1 + EDIST(A[1..(m-1)], B[1..(n-1)])$
    return $\min(m_1, m_2, m_3)$
```

Example

DEED and DREAD

Subproblems and Recurrence

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

Optimal Costs

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

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

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

Memoizing the Recursive Algorithm

```
$int \quad M[0..m][0..n]$
Initialize all entries of $M[i][j]$ to $\infty$
return $EDIST(A[1..m], B[1..n])$
```

```
$EDIST(A[1..m], B[1..n])$
    If $(M[i][j] < \infty)$ return $M[i][j]$   (* return stored value *)
    If $(m = 0)$
        $M[i][j] = n$
    ElseIf $(n = 0)$
        $M[i][j] = m$
    Else
        $m_1 = 1 + EDIST(A[1..(m-1)], B[1..n])$
        $m_2 = 1 + EDIST(A[1..m], B[1..(n-1)]))$
        If $(A[m] = B[n])$ $m_3 = EDIST(A[1..(m-1)], B[1..(n-1)])$
        Else $m_3 = 1 + EDIST(A[1..(m-1)], B[1..(n-1)])$
        $M[i][j] = \min(m_1, m_2, m_3)$
    return $M[i][j]$
```

Removing Recursion to obtain Iterative Algorithm

```
$EDIST(A[1..m], B[1..n])$
    int  $M[0..m][0..n]$
    for $i = 1$ to $m$ do $M[i, 0] = i$
    for $j = 1$ to $n$ do $M[0, j] = j$

    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            $M[i][j] = \min \begin{cases} [x_i = y_j] + M[i-1][j-1], \\ 1 + M[i-1][j], \\ 1 + M[i][j-1] \end{cases}$
```

Removing Recursion to obtain Iterative Algorithm

```
$EDIST(A[1..m], B[1..n])$
    int  $M[0..m][0..n]$
    for $i = 1$ to $m$ do $M[i, 0] = i$
    for $j = 1$ to $n$ do $M[0, j] = j$

    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            $M[i][j] = \min \begin{cases} [x_i = y_j] + M[i-1][j-1], \\ 1 + M[i-1][j], \\ 1 + M[i][j-1] \end{cases}$
```

Analysis

Removing Recursion to obtain Iterative Algorithm

```
$EDIST(A[1..m], B[1..n])$
    $int \quad M[0..m][0..n]$
    for $i = 1$ to $m$ do $M[i, 0] = i$
    for $j = 1$ to $n$ do $M[0, j] = j$

    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            $M[i][j] = \min \begin{cases} [x_i = y_j] + M[i-1][j-1], \\ 1 + M[i-1][j], \\ 1 + M[i][j-1] \end{cases}$
```

Analysis

1. Running time is $O(mn)$.
2. Space used is $O(mn)$.

Matrix and DAG of Computation

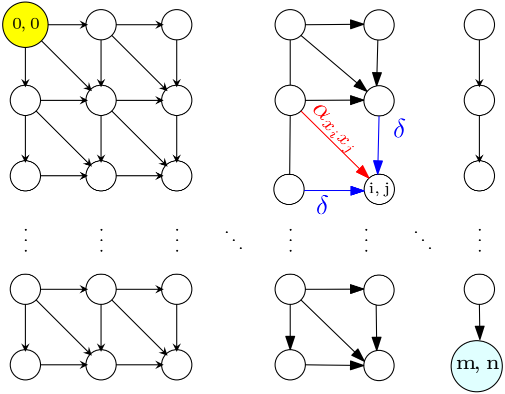


Figure: Iterative algorithm in previous slide computes values in row order.

Finding an Optimum Solution

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

Question: Can we find a specific alignment which achieves the minimum?

Finding an Optimum Solution

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

Question: Can we find a specific alignment which achieves the minimum?

Exercise: Show that one can find an optimum solution after computing the optimum value. Key idea is to store back pointers when computing $Opt(i, j)$ to know how we calculated it. See notes for more details.

Longest Palindromic Subsequence

Definition

A sequence is a *palindrome* if the sequence is equal to its reverse.
Examples: m,a,l,a,y,a,l,a,m and 1,10,10,1 and a.

Longest Palindromic Subsequence

Definition

A sequence is a *palindrome* if the sequence is equal to its reverse.
Examples: m,a,l,a,y,a,l,a,m and 1,10,10,1 and a.

Problem: Given a sequence $a_0, a_1, \ldots, a_n$ find the *longest* palindromic sub-sequence.

Examples:

- $1, 10, 11$
- a, c, c, r, a
- $A, C, G, T, G, T, C, A, A, A, A, T, C, G$

Dynamic Programming Template

1. Come up with a recursive algorithm to solve problem
2. Understand the structure/number of the subproblems generated by recursion
3. Memoize the recursion
    - set up compact notation for subproblems
    - set up a data structure for storing subproblems
4. Iterative algorithm
    - Understand dependency graph on subproblems
    - Pick an evaluation order (any topological sort of the dependency dag)
5. Analyze time and space
6. Optimize