Write a (very simple) recursive algorithm that calculates the Fibonacci $n^{th}$ number.

$$F_n = F_{n-1} + F_{n-2} \text{ where } F_0 = 0, F_1 = 1$$
Write a (very simple) recursive algorithm that calculates the Fibonacci $n^{th}$ number.

$$F_n = F_{n-1} + F_{n-2} \text{ where } F_0 = 0, F_1 = 1$$
Recursion and Memoization
Fibonacci Numbers

Fibonacci numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \text{ and } F(0) = 0, F(1) = 1. \]

These numbers have many interesting properties. A journal The Fibonacci Quarterly! is the golden ratio \((\frac{1}{2}(1 + \sqrt{5}))\) or approximately 1.618.\]
Fibonacci Numbers

Fibonacci numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \] and \[ F(0) = 0, \, F(1) = 1. \]

These numbers have many interesting properties. A journal **The Fibonacci Quarterly**!  

- **Binet’s formula**: \[ F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \approx \frac{1.618^n - (-0.618)^n}{\sqrt{5}} \approx \frac{1.618^n}{\sqrt{5}} \]  
\( \varphi \) is the golden ratio \((1 + \sqrt{5})/2 \approx 1.618. \)

- \( \lim_{n \to \infty} F(n + 1)/F(n) = \varphi \)
Recursive Algorithm for Fibonacci Numbers

**Question:** Given $n$, compute $F(n)$.

\[
\text{Fib}(n):
\begin{align*}
\text{if } (n = 0) & \quad \text{return } 0 \\
\text{else if } (n = 1) & \quad \text{return } 1 \\
\text{else} & \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]
Recursive Algorithm for Fibonacci Numbers

**Question:** Given $n$, compute $F(n)$.

```python
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n - 1) + Fib(n - 2)
```

Running time? Let $T(n)$ be the number of additions in Fib(n).

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

$Fib = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

$m = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots$
Recursive Algorithm for Fibonacci Numbers

**Question:** Given $n$, compute $F(n)$.

$$\text{Fib}(n):$$

```python
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n - 1) + Fib(n - 2)
```

Running time? Let $T(n)$ be the number of additions in Fib(n).

$$T(n) = T(n - 1) + T(n - 2) + 1 \text{ and } T(0) = T(1) = 0$$
Recursive Algorithm for Fibonacci Numbers

**Question:** Given $n$, compute $F(n)$.

```python
Fib(n):
    if ($n = 0$)
        return 0
    else if ($n = 1$)
        return 1
    else
        return $Fib(n - 1) + Fib(n - 2)$
```

Running time? Let $T(n)$ be the number of additions in $Fib(n)$.

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

$T(0) = T(1) = 0$

$$T(n) = O(2^n)$$

Roughly same as $F(n)$: $T(n) = \Theta(\varphi^n) < O(2^n)$

The number of additions is exponential in $n$. Can we do better?
Recursion tree for the Recursive Fibonacci

0 1
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci

0  1
2  1
0  1
3  2
0  1
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci
An iterative algorithm for Fibonacci numbers

\[ \text{FibIter}(n): \]
\begin{align*}
\text{if } (n = 0) & \text{ then} \\
& \quad \text{return } 0 \\
\text{if } (n = 1) & \text{ then} \\
& \quad \text{return } 1 \\
F[0] &= 0 \\
F[1] &= 1 \\
\text{for } i = 2 \text{ to } n & \text{ do} \\
& \quad F[i] = F[i - 1] + F[i - 2] \\
\text{return } F[n]
\end{align*}

What is the running time of the algorithm? \( O(n) \) additions.

\[ F = [0, 1, 1, 2, \ldots \ldots] \]
An iterative algorithm for Fibonacci numbers

FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i - 1] + F[i - 2]
    return F[n]

What is the running time of the algorithm?

\( O(n) \cdot O(1) = O(n) \)
An iterative algorithm for Fibonacci numbers

FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i-1] + F[i-2]
    return F[n]

What is the running time of the algorithm? \( O(n) \) additions.
What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value.
What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. **Memoization**.
What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming: Finding a recursion that can be effectively/efficiently memorized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.
Automatic/implicit memorization
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```python
def Fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if Fib(n) was previously computed:
        return stored value of Fib(n)
    else:
        return Fib(n-1) + Fib(n-2)
```

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure).
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n) : \\
\begin{align*}
\text{if} & \quad (n = 0) \\
& \quad \text{return} \ 0 \\
\text{if} & \quad (n = 1) \\
& \quad \text{return} \ 1 \\
\text{if} & \quad (\text{Fib}(n) \text{ was previously computed}) \\
& \quad \text{return} \ \text{stored value of Fib}(n) \\
\text{else} & \\
& \quad \text{return} \ \text{Fib}(n - 1) + \ \text{Fib}(n - 2)
\end{align*}
\]
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n): \\
\quad \text{if } (n = 0) \\
\quad \quad \text{return } 0 \\
\quad \text{if } (n = 1) \\
\quad \quad \text{return } 1 \\
\quad \text{if } (\text{Fib}(n) \text{ was previously computed}) \\
\quad \quad \text{return stored value of Fib(n)} \\
\quad \text{else} \\
\quad \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\]

How do we keep track of previously computed values?
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n):
\]
\[
\text{if } (n = 0) \quad \text{return } 0 \\
\text{if } (n = 1) \quad \text{return } 1 \\
\text{if } (\text{Fib}(n) \text{ was previously computed}) \quad \text{return stored value of Fib}(n) \\
\text{else} \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\]

How do we keep track of previously computed values? Two methods: explicitly and implicitly (via data structure)
Automatic implicit memorization

Initialize a (dynamic) dictionary data structure $D$ to empty

$$\text{Fib}(n):$$

if $(n = 0)$
    return 0

if $(n = 1)$
    return 1

if $(n$ is already in $D$)
    return value stored with $n$ in $D$

val $\leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2)$

Store $(n, \text{val})$ in $D$

return val

Use hash-table or a map to remember which values were already computed.
Explicit memorization (not automatic)

- Initialize table/array $M$ of size $n$: $M[i] = -1$ for $i = 0, \ldots, n$. 

\[
\begin{align*}
\text{Fib}(n) &= \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
M[n] & \text{if } M[n] \neq -1 \\
M[n] \left( \text{Fib}(n-1) + \text{Fib}(n-2) \right) & \text{otherwise}
\end{cases}
\end{align*}
\]

- Need to know upfront the number of sub-problems to allocate memory.
Explicit memorization (not automatic)

- Initialize table/array $M$ of size $n$: $M[i] = -1$ for $i = 0, \ldots, n$.
- Resulting code:

```python
Fib(n):
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    if (M[n] != -1) // M[n]: stored value of Fib(n)
        return M[n]
    M[n] ← Fib(n - 1) + Fib(n - 2)
    return M[n]
```
Explicit memorization (not automatic)

- Initialize table/array $M$ of size $n$: $M[i] = -1$ for $i = 0, \ldots, n$.
- Resulting code:
  
  ```python
  def Fib(n):
      if (n == 0):
          return 0
      if (n == 1):
          return 1
      if (M[n] != -1) // M[n]: stored value of Fib(n)
          return M[n]
      M[n] = Fib(n - 1) + Fib(n - 2)
      return M[n]
  ```

- Need to know upfront the number of sub-problems to allocate memory.
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Recursion tree for the memorized Fib...
Automatic (Implicit) Memorization

- Recursive version:

  \[ f(x_1, x_2, \ldots, x_d) : \]
  
  CODE

- Recursive version with memoization:

  \[ g(x_1, x_2, \ldots, x_d) : \]
  
  ```
  if f already computed for (x_1, x_2, \ldots, x_d) then
  return value already computed
  ```
  
  NEW_CODE

- NEW_CODE:
  - Replaces any “return \( \alpha \)” with
  - Remember “\( f(x_1, \ldots, x_d) = \alpha \)” ; return \( \alpha \).
Explicit vs Implicit Memoization

- Explicit memoization (on the way to iterative algorithm) preferred:
  - analyze problem ahead of time

- Implicit (automatic) memoization:
  - problem structure or algorithm is not well understood.
  - Need to pay overhead of data-structure.

Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.
Explicit vs Implicit Memoization

- Explicit memoization (on the way to iterative algorithm) preferred:
  - analyze problem ahead of time
  - Allows for efficient memory allocation and access.
Explicit vs Implicit Memoization

- Explicit memoization (on the way to iterative algorithm) preferred:
  - analyze problem ahead of time
  - Allows for efficient memory allocation and access.
- Implicit (automatic) memoization:
  - problem structure or algorithm is not well understood.
Explicit vs Implicit Memoization

- Explicit memoization (on the way to iterative algorithm) preferred:
  - analyze problem ahead of time
  - Allows for efficient memory allocation and access.

- Implicit (automatic) memoization:
  - problem structure or algorithm is not well understood.
  - Need to pay overhead of data-structure.
Explicit vs Implicit Memoization

- Explicit memoization (on the way to iterative algorithm) preferred:
  - analyze problem ahead of time
  - Allows for efficient memory allocation and access.

- Implicit (automatic) memoization:
  - problem structure or algorithm is not well understood.
  - Need to pay overhead of data-structure.
  - Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.
Explicit/implicit memorization for Fibonacci

Init: \( M[i] = -1, \ i = 0, \ldots, n. \)

**Fib\( (k)\):**
- if \( (k = 0) \)
  return 0
- if \( (k = 1) \)
  return 1
- if \( (M[k] \neq -1) \)
  return \( M[n] \)

\( M[k] \leftarrow \text{Fib}(k - 1) + \text{Fib}(k - 2) \)
return \( M[k] \)

Init: Init dictionary \( D \)

**Fib\( (n)\):**
- if \( (n = 0) \)
  return 0
- if \( (n = 1) \)
  return 1
- if \( (n \text{ is already in } D) \)
  return value stored with \( n \) in \( D \)
  \( \text{val} \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2) \)
Store \( (n, \text{val}) \) in \( D \)
return \( \text{val} \)

Explicit memorization

Implicit memorization
Dynamic programming
Removing the recursion by filling the table in the right order

\begin{align*}
\text{Fib}(n): & \quad \text{if } (n = 0) \quad \text{return } 0 \\
& \quad \text{if } (n = 1) \quad \text{return } 1 \\
& \quad \text{if } (M[n] \neq -1) \quad \text{return } M[n] \\
& \quad M[n] \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2) \\
& \quad \text{return } M[n]
\end{align*}

\begin{align*}
\text{FibIter}(n): & \quad \text{if } (n = 0) \quad \text{then} \\
& \quad \text{return } 0 \\
& \quad \text{if } (n = 1) \quad \text{then} \\
& \quad \text{return } 1 \\
& \quad F[0] = 0 \\
& \quad F[1] = 1 \\
& \quad \text{for } i = 2 \text{ to } n \text{ do} \\
& \quad \quad F[i] = F[i - 1] + F[i - 2] \\
& \quad \text{return } F[n]
\end{align*}
Dynamic programming: Saving space!

Saving space. Do we need an array of \( n \) numbers? Not really.

\[
\text{FibIter}(n):
\]

\[
\begin{align*}
&\text{if } (n = 0) \text{ then} \\
&\quad \text{return } 0 \\
&\text{if } (n = 1) \text{ then} \\
&\quad \text{return } 1 \\
&F[0] = 0 \\
&F[1] = 1 \\
&\text{for } i = 2 \text{ to } n \text{ do} \\
&\quad F[i] = F[i - 1] + F[i - 2] \\
&\text{return } F[n]
\end{align*}
\]

\[
\text{FibIter}(n):
\]

\[
\begin{align*}
&\text{if } (n = 0) \text{ then} \\
&\quad \text{return } 0 \\
&\text{if } (n = 1) \text{ then} \\
&\quad \text{return } 1 \\
&prev2 = 0 \\
&prev1 = 1 \\
&\text{for } i = 2 \text{ to } n \text{ do} \\
&\quad \text{temp} = prev1 + prev2 \\
&\quad prev2 = prev1 \\
&\quad prev1 = \text{temp} \\
&\text{return } prev1
\end{align*}
\]
Dynamic programming – quick review

Dynamic Programming is **smart recursion**
Dynamic Programming is \textbf{smart recursion} + \textbf{explicit memorization}
Dynamic Programming is **smart recursion**

+ **explicit memorization**
+ filling the table in right order
+ removing recursion.
Suppose we have a recursive program $foo(x)$ that takes an input $x$.

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

Suppose we memorize the recursion.

Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

Q: What is an upper bound on the running time of the memorized version of $foo(x)$ if $|x| = n$?

$O(A(n)B(n))$. 

\[ F_b: A(n) = n \]

\[ B(n) = O(n) \]

(1) addition
Analyzing memorized recursive function

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.

Suppose we memorize the recursion.
**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.
Analyzing memorized recursive function

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.

Suppose we memorize the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

**Q:** What is an upper bound on the running time of memorized version of $foo(x)$ if $|x| = n$? $A(n) \cdot B(n)$
Analyzing memorized recursive function

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.

Suppose we **memorize** the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

Q: What is an upper bound on the running time of **memorized** version of $foo(x)$ if $|x| = n$? $O(A(n)B(n))$. 
Fibonacci numbers are big – corrected running time analysis
Is the iterative algorithm a polynomial time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
Longest Increasing Sub-sequence Revisited
**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 **sub-sequence** 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**.
Example

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

**Input**  A sequence of numbers \( a_0, a_1, \ldots, a_{n-1} \)

**Goal**  Find an increasing subsequence \( a_{i_0}, a_{i_1}, \ldots, a_{i_k} \) of maximum length
Longest Increasing Subsequence Problem

**Input**  A sequence of numbers $a_0, a_1, \ldots, a_{n-1}$

**Goal**  Find an increasing subsequence $a_{i_0}, a_{i_1}, \ldots, a_{i_k}$ of maximum length

**Example**
- Sequence: 6, 3, 5, 2, 7, 8, 1
- Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
- Longest increasing subsequence: 3, 5, 7, 8
- This is just for [6,3,5,2,7]! (Tikz won’t print larger trees)
- How many leaves are there for the full [6,3,5,2,7, 8, 1] sequence
- What is the running time?
Naive Recursion Enumeration - Code

Assume $a_1, a_2, \ldots, a_n$ is contained in an array $A$

```python
algLISNaive(A[1..n]):
    max = 0
    for each subsequence $B$ of $A$
do
        if $B$ is increasing and $|B| > max$ then
            max = $|B|$ 

Output max
```

**Running time:** $O(n2^n)$.

$2^n$ subsequences of a sequence of length $n$ and $O(n)$ time to check if a given sequence is increasing.
Can we find a recursive algorithm for LIS?

\textbf{LIS}(A[0..'n - 1]):
Can we find a recursive algorithm for LIS?

**LIS**\( (A[0..n-1]) \):

- **Case 1**: Does not contain \( A[n-1] \) in which case 
  \[ \text{LIS}(A[0..n-1]) = \text{LIS}(A[0..(n-1)]) \]

- **Case 2**: contains \( A[n-1] \) in which case \( \text{LIS}(A[0..n-1]) \) is not so clear.

**Observation**
*For second case we want to find a subsequence in \( A[1..(n-2)] \) that is restricted to numbers less than \( A[n-1] \). This suggests that a more general problem is **LIS_smaller**(\( A[0..n-1], x \)) which gives the longest increasing subsequence in \( A \) where each number in the sequence is less than \( x \).*
Sequence: $A[0..6] = 6, 3, 5, 2, 7, 8, 1$
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..i], x):
    if $i = 0$ then return 0
    $m = LIS_{smaller}(A[1..i - 1], x)$
    if $A[i] < x$ then
        $m = max(m, 1 + LIS_{smaller}(A[1..i - 1], A[i]))$
    Output $m$
```

$LIS(A[1..n])$:
    return $LIS_{smaller}(A[1..n], \infty)$
Recursive Approach

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

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

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

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

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

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

\[
\text{LIS\_smaller}(A[1..i], x) : \\
\hspace{1em} \text{if } i = 0 \text{ then return } 0 \\
\hspace{1em} m = \text{LIS\_smaller}(A[1..i - 1], x) \\
\hspace{1em} \text{if } A[i] < x \text{ then} \\
\hspace{2em} m = \max(m, 1 + \text{LIS\_smaller}(A[1..i - 1], A[i])) \\
\hspace{1em} \text{Output } m
\]

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

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

\[
A(n) = O(n^2) \\
B(n) = O(1)
\]
Recursive Approach

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

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

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

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

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

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

\[
\text{LIS\_smaller}(A[1..i], x):
\]
\begin{enumerate}
  \item if \( i = 0 \) then return 0
  \item \( m = \text{LIS\_smaller}(A[1..i - 1], x) \)
  \item if \( A[i] < x \) then
    \begin{enumerate}
      \item \( m = \max(m, 1 + \text{LIS\_smaller}(A[1..i - 1], A[i])) \)
    \end{enumerate}
\end{enumerate}
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 memorize 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 memorization? \( O(n^2) \)
Naming sub-problems and recursive equation

After seeing that number of sub-problems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add $\infty$ at end of array (in position $n + 1$)

$LIS(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)
Naming sub-problems and recursive equation

After seeing that number of sub-problems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add $\infty$ at end of array (in position $n+1$)

$LIS(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

**Base case:** $LIS(0, j) = 0$ for $1 \leq j \leq n + 1$

**Recursive relation:**
- $LIS(i, j) = LIS(i - 1, j)$ if $A[i] \geq A[j]$
- $LIS(i, j) = \max \{LIS(i - 1, j), 1 + LIS(i - 1, i)\}$ if $A[i] < A[j]$

**Output:** $LIS(n, n + 1)$. 
How to order bottom up computation?

Sequence:

\[ A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \]

Recursive relation:

\[ LIS(i, j) = \begin{cases} 
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases} \]
How to order bottom up computation?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

 Represents limiter

\[ j \]

\[ i \]

\[ \text{Represents sub-array} \]

Sequence:
\[ A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \]

Recursive relation:

\[ LIS(i, j) = \]

\[ \begin{cases} 
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases} \]
How to order bottom up computation?

Sequence:

\[ A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \]

Recursive relation:

\[
LIS(i, j) = \begin{cases} 
  0 & i = 0 \\
  LIS(i - 1, j) & A[i] \geq A[j] \\
  \max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases}
\]
How to order bottom up computation?

Sequence:
\[ A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \]

Recursive relation:
\[
LIS(i, j) =
\begin{cases}
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases}
\]
How to order bottom up computation?

Sequence:
\[ A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \]

Recursive relation:
\[
LIS(i, j) = \begin{cases} 
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases}
\]
# How to order bottom up computation?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[6]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[6,3]</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[6,3,5]</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[6,3,5,2]</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[6,3,5,2,7]</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[6,3,5,2,7,8]</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[6,3,5,2,7,8,1]</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Represents limiter \( j \)

Represents sub-array \( i \)

## Recursive relation:

\[
LIS(i, j) = \begin{cases} 
0 & \text{if } i = 0 \\
LIS(i - 1, j) & \text{if } A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & \text{if } A[i] < A[j]
\end{cases}
\]

## Sequence:

\( A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \)
How to order bottom up computation?

Sequence:
\[ A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \]

Recursive relation:

\[
LIS(i, j) = \begin{cases} 
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases}
\]
How to order bottom up computation?

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Represents limiter</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3]</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3,5]</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2]</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2,7]</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2,7,8]</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2,7,8,1]</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Represents sub-array i

Recursive relation:

\[
LIS(i, j) = \begin{cases} 
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases}
\]

Sequence:

\[A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1]\]
**How to order bottom up computation?**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[6]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[6,3]</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>[6,3,5]</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>[6,3,5,2]</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>[6,3,5,2,7]</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>[6,3,5,2,7,8]</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>[6,3,5,2,7,8,1]</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Receives a sub-array \( i \)

Receives a limiter \( j \)

**Recursive relation:**

\[
LIS(i, j) =
\begin{cases} 
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ \begin{array}{l}
LIS(i - 1, j) \\
1 + LIS(i - 1, i)
\end{array} \right\} & A[i] < A[j]
\end{cases}
\]

**Sequence:**

\[
A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1]
\]
Iterative algorithm

The dynamic program for longest increasing subsequence

\[ \text{LIS-Iterative}(A[1..n]) : \]
\[
A[n + 1] = \infty
\]
\[
\text{int } \text{LIS}[0..n - 1, 0..n]
\]
\[
\text{for } j = 0 \ldots n \) \text{ if } A[i] \leq A[j] \text{ then } \text{LIS}[0][j] = 1
\]
\[
\text{for } i = 1 \ldots n - 1 \text{ do }
\]
\[
\text{for } j = i \ldots n - 1 \text{ do }
\]
\[
\text{if } (A[i] \geq A[j])
\]
\[
\text{LIS}[i, j] = \text{LIS}[i - 1, j]
\]
\[
\text{else}
\]
\[
\text{LIS}[i, j] = \max(\text{LIS}[i - 1, j], 1 + \text{LIS}[i - 1, i])
\]
\[
\text{Return } \text{LIS}[n, n + 1]
\]

Running time: \( O(n^2) \)
Space: \( O(n^2) \)
Iterative algorithm

The dynamic program for longest increasing subsequence

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

- \( A[n + 1] = \infty \)
- \( \text{int } LIS[0..n - 1, 0..n] \)
- \( \text{for } j = 0 \ldots n \text{ do if } A[i] \leq A[j] \text{ then } LIS[0][j] = 1 \)

- \( \text{for } i = 1 \ldots n - 1 \text{ do} \)
  - \( \text{for } j = i \ldots n - 1 \text{ do} \)
    - \( \text{if } (A[i] \geq A[j]) \)
      - \( LIS[i, j] = LIS[i - 1, j] \)
    - \( \text{else} \)
      - \( LIS[i, j] = \max(LIS[i - 1, j], 1 + LIS[i - 1, i]) \)

- Return \( LIS[n, n + 1] \)

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

Space: \( O(n^2) \) Can be done in linear space. How?
Two comments

Question: Can we compute an optimum solution and not just its value?
Question: Can we compute an optimum solution and not just its value?
Yes! See notes.
Finding the sub-sequence

Sequence:
\[ A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1] \]

We know the LIS length (4) but how do we find the LIS itself?

\[ LIS = [3, 5, 7, 8] \]

\[ LIS(i, j) = \begin{cases} 
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases} \]
**Finding the sub-sequence**

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>[6]</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[6,3]</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>[6,3,5]</td>
<td>3</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2]</td>
<td>4</td>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2,7]</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2,7,8]</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3,5,2,7,8,1]</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Represents sub-array i

Represents limiter j

Sequence:

A[1 \ldots 7] = [6, 3, 5, 2, 7, 8, 1]

We know the LIS length (4) but how do we find the LIS itself?

**Recursive relation:**

\[
LIS(i, j) =
\begin{cases}
0 & i = 0 \\
LIS(i - 1, j) & A[i] \geq A[j] \\
\max \left\{ LIS(i - 1, j), 1 + LIS(i - 1, i) \right\} & A[i] < A[j]
\end{cases}
\]
Two comments

**Question:** Can we compute an optimum solution and not just its value?
Yes!

**Question:** Is there a faster algorithm for LIS?
Two comments

**Question:** Can we compute an optimum solution and not just its value?
Yes!

**Question:** Is there a faster algorithm for LIS? Yes! Using a different recursion and optimizing one can obtain an $O(n \log n)$ time and $O(n)$ space algorithm. $O(n \log n)$ time is not obvious. Depends on improving time by using data structures on top of dynamic programming.
How to come up with dynamic programming algorithm: summary
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- This gives an upper bound on the total running time if we use automatic/explicit memorization.

...need to find the right way or order the sub-problems evaluation. This leads to an a dynamic programming algorithm.

Optimize the resulting algorithm further...
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- This gives an upper bound on the total running time if we use automatic/explicit memorization.
- Come up with an explicit memorization algorithm for the problem.
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- This gives an upper bound on the total running time if we use automatic/explicit memorization.
- Come up with an explicit memorization algorithm for the problem.
- Eliminate recursion and find an iterative algorithm.
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- This gives an upper bound on the total running time if we use automatic/explicit memorization.
- Come up with an explicit memorization algorithm for the problem.
- Eliminate recursion and find an iterative algorithm.
- ...need to find the right way or order the sub-problems evaluation.
  This leads to an a dynamic programming algorithm.
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- This gives an upper bound on the total running time if we use automatic/explicit memorization.
- Come up with an explicit memorization algorithm for the problem.
- Eliminate recursion and find an iterative algorithm.
- ...need to find the right way or order the sub-problems evaluation. This leads to an a dynamic programming algorithm.
- Optimize the resulting algorithm further
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- This gives an upper bound on the total running time if we use automatic/explicit memorization.
- Come up with an explicit memorization algorithm for the problem.
- Eliminate recursion and find an iterative algorithm.
- ...need to find the right way or order the sub-problems evaluation. This leads to an a dynamic programming algorithm.
- Optimize the resulting algorithm further
- ...
Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- This gives an upper bound on the total running time if we use automatic/explicit memorization.
- Come up with an explicit memorization algorithm for the problem.
- Eliminate recursion and find an iterative algorithm.
- ...need to find the right way or order the sub-problems evaluation. This leads to an a dynamic programming algorithm.
- Optimize the resulting algorithm further
- ...
- Get rich!