**Fibonacci**

- Fibonacci Numbers (circa 13th century)

<table>
<thead>
<tr>
<th>$F_n$ =</th>
<th>0 if $n$ = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 if $n$ = 1</td>
</tr>
<tr>
<td></td>
<td>$F_{n-1} + F_{n-2}$ o/w</td>
</tr>
</tbody>
</table>

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

- Translates line by line to code:

```python
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:

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

Running time? (backtracking recurrence)

\[
T(n) = T(n-1) + T(n-2) + O(1) = \Theta(F_n) = \Theta(1.618^n) = \Theta(((\sqrt{5}+1)/2)^n)
\]
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!
How many intermediate nodes does a full binary tree with m leaves have?
Running time via Rec Tree

\[ 2F_{n+1} - 1 \text{ nodes (additions)} \]
Running time via Rec Tree
Running time via Rec Tree

Keep an array to remember the previous values!
Running time via Rec Tree

F_5

F_4  F_3
  F_2  F_2
   F_1  F_1
    F_0  F_0
     F_0  F_0

| 0 | 1 | 2 | 3 | 4 | 5 | ...
|---|---|---|---|---|---|---
| 0 | 1 | 1 | 2 |   |   |   |
Running time via Rec Tree

look up array for F_2

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>...</td>
</tr>
</tbody>
</table>
Running time via Rec Tree

look up array for $F_3$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Running time via Rec Tree

Memoization= when I look at the table to see the values I computed before
Given any recursive backtracking algorithm, you can add memoization and will save time, provided the subproblems repeat.
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
Memoized algorithm fills in the table from left to right. Why not just do that?
Memoized algorithm fills in the table from left to right. Why not just do that?

We get an iterative algorithm

\[
\text{IterFibo}(n):
\begin{align*}
F[0] & \leftarrow 0 \\
F[1] & \leftarrow 1 \\
\text{for } i & \leftarrow 2 \text{ to } n \\
F[i] & \leftarrow F[i-1] + F[i-2] \\
\text{return } F[n]
\end{align*}
\]
• 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):

\[
\begin{align*}
F[0] & \leftarrow 0 \\
F[1] & \leftarrow 1 \\
\text{for } i & \leftarrow 2 \text{ to } n \\
F[i] & \leftarrow F[i-1] + F[i-2] \\
\text{return } F[n]
\end{align*}
\]

- Structure mirrors the recurrence
- Only subtle thing is that we want to fill in the array in increasing 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] ← 0
    F[1] ← 1
    for i ← 2 to n
       F[i] ← 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?

```plaintext
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
```

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

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

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(log\(n\)) multiplications
• Compute \(F_n\) in O(log\(n\)) 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)

- \[ 31415926538279461048 \]

- \[ \text{LIS}(A[1...n], p) = \begin{cases} 
0 & \text{if } n = 0 \\
\text{LIS}(A[2...n], p) & \text{if } A[1] \leq p \\
\max \{ \text{LIS}(A[2...n], p), 1 + \text{LIS}(A[2...n], A[1]) \} & \end{cases} \]
Longest Increasing Subsequence (LIS)

• \( \text{LIS}(A[1...n], p) = \begin{cases} 
0 & \text{if } n = 0 \\
\text{LIS}(A[2...n], p) & \text{if } A[1] \leq p \\
\max \{ \text{LIS}(A[2...n], p), 1 + \text{LIS}(A[2...n], A[1]) \} & \text{otherwise}
\end{cases} \)

• The argument \( p \) is always either \(-\infty\) or an element of the array \( A \)
• Add \( A[0] = -\infty \)
• We can identify any recursive subproblem with two array indices.
• \( \text{LIS}(i, j) = \text{length or LIS of } A[j...n] \text{ with all elements larger than } A[i] \)
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 than $A[i]$
- We want to compute $LIS(0,1)$
- Memoize? what data structure to use?
- Two dimensional Array $LIS[0…n,1…n+1]$
For $i < j$

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>j</th>
<th></th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Purple squares must be filled before pink
For $i < j$

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

don’t matter what order I fill the columns in

```
LIS(A[1..n]):
  A[0] ← -∞  ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n ⟨⟨Base cases⟩⟩
    LIS[i, n + 1] ← 0
  for j ← n downto 1
    for i ← 0 to j - 1
      if A[i] ≥ A[j]
        LIS[i, j] ← LIS[i, j + 1]
      else
        LIS[i, j] ← 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 many values are there in the recurrence?

```
LIS(A[1..n]):
    A[0] ← −∞  ⟨Add a sentinel⟩
    for i ← 0 to n  ⟨Base cases⟩
        LIS[i, n + 1] ← 0
    for j ← n downto 1
        for i ← 0 to j − 1
            if A[i] ≥ A[j]
                LIS[i, j] ← LIS[i, j + 1]
            else
                LIS[i, j] ← 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