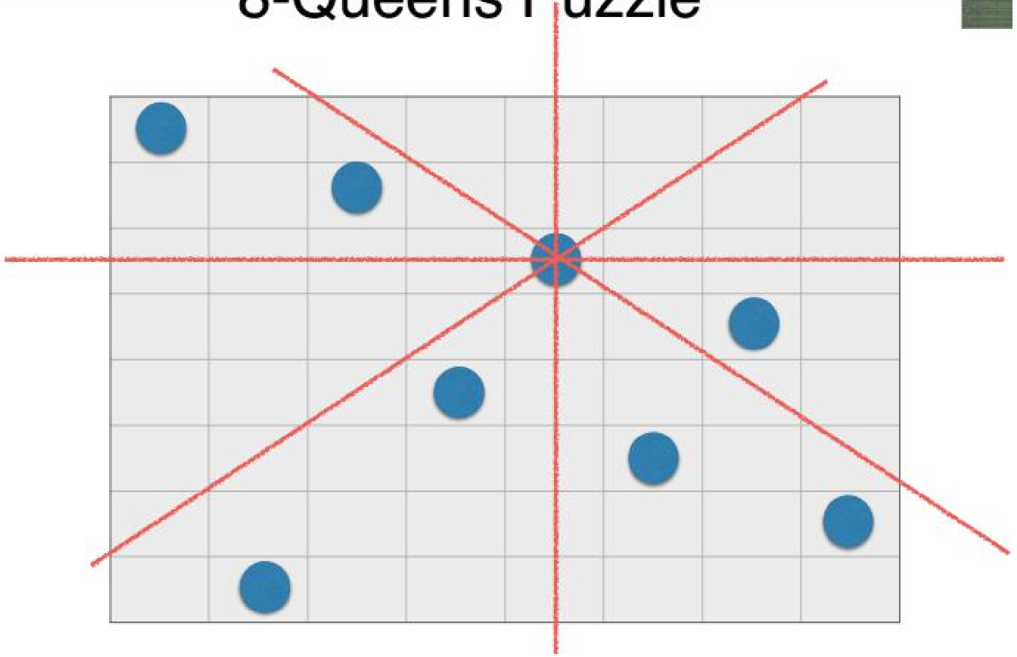
8-Queens Puzzle



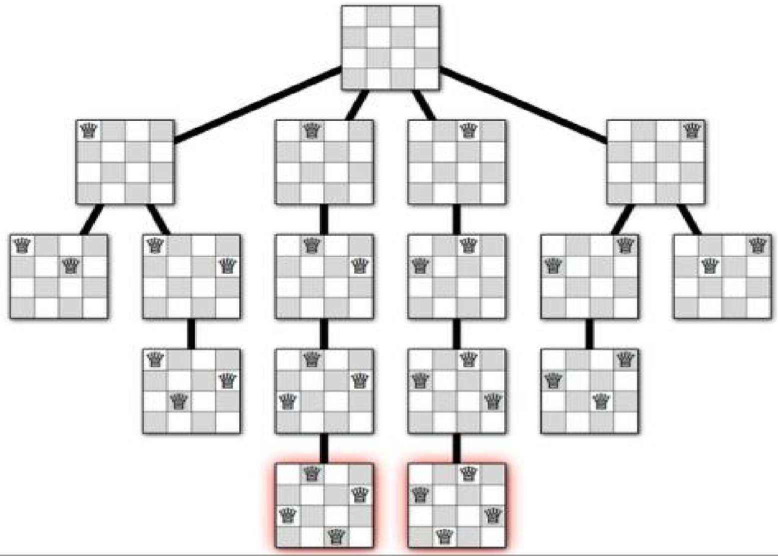
n-Queens Puzzle

Place a queen at the first empty row-try all possible places

n-Queens Puzzle

```
RecursiveNQueens(Q[1..n], r):
  if r = n + 1
      print Q
  else
      for j ← 1 to n
          legal ← True
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← False
          if legal
              Q[r] ← j
              RecursiveNQueens(Q[1..n], r + 1)
```

n-Queens Puzzle



Subset sum

- Given a set X of positive integers and a target positive integer t, is there a subset of elements in X that add up to t?
- Given X, find A subset of X, so that ∑A=t?
- What is the first element to go into A?
- Try them all!
- If there is an element equal to t, done
- If t is zero, we are done! (why?)
- If t negative, no!

Subset sum

- Given a set X of positive integers and a target positive integer t, is there a subset of elements in X that add up to t?
- Given X, find A subset of X, so that ∑A=t?
- Example: X={3,2,4,6,9}, t = 7
- What element to try first?
- Say x= 6. Then is there subset of {3,2,4,9} that adds to 1? NO
- Two cases: x in A or x not in A.

Subset sum

- If there is a subset A with ∑A=t then either
- x in A, call SubsetSum(X-{x},t-x)
- or x not in A call SubsetSum(X-{x},t)

Subset sum

```
SubsetSum(X[1..n], T):
  if T = 0
      return True
  else if T < 0 or n = 0
      return False
  else
      return (SubsetSum(X[1..n−1], T) ∨ SubsetSum(X[1..n−1], T − X[n]))
```

Call the algorithm with i=n
Canonical order to choose elements in the subset

Longest Increasing Subsequence (LIS)

- 3 1 4 1 5 9 2 6 5 3 8 2 7 9 4 6 1 0 4 8
- Subsequence different than substring.
- Increasing = in an order.
- Recursion?

Longest Increasing Subsequence (LIS)

- 3 1 4 1 5 9 2 6 5 3 8 2 7 9 4 6 1 0 4 8
- Look at first element. Keep or ditch?
   - LIS(A[1…n])

If n< 10^{10}, brute force

keep: 1+LIS(A[2…n])

ditch: LIS(A[2…n])

What went wrong?
I didn't use
INCREASING

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)

If n< 10^{10}, brute force

If A[1] $\leq$ p,

RETURN LIS(A[2…n],p)

else

RETURN MAX: LIS(A[2…n],p)
1+LIS(A[2…n],A[1])

Recursion

Reduction:

Reduce one problem to another

Recursion

A special case of reduction

1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

1. Problem instance of size **n** is reduced to one or more instances of size **n − 1** or less.
2. For termination, problem instances of small size are solved by some other method as **base cases**.

Recursion in Algorithm Design

1. **Tail Recursion**: problem reduced to a *single* recursive call after some work. Easy to convert algorithm into iterative or greedy algorithms. Examples: Interval scheduling, MST algorithms, etc.
2. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems that are solved separately. Conquer step puts together solution for bigger problem.

   Examples: Closest pair, deterministic median selection, quick sort.
3. **Backtracking**: Refinement of brute force search. Build solution incrementally by invoking recursion to try all possibilities for the decision in each step.
4. **Dynamic Programming**: problem reduced to multiple (typically) *dependent or overlapping* sub-problems. Use **memoization** to avoid recomputation of common solutions leading to *iterative bottom-up* algorithm.

Longest Increasing Subsequence Problem

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

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

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1, 9
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8

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:

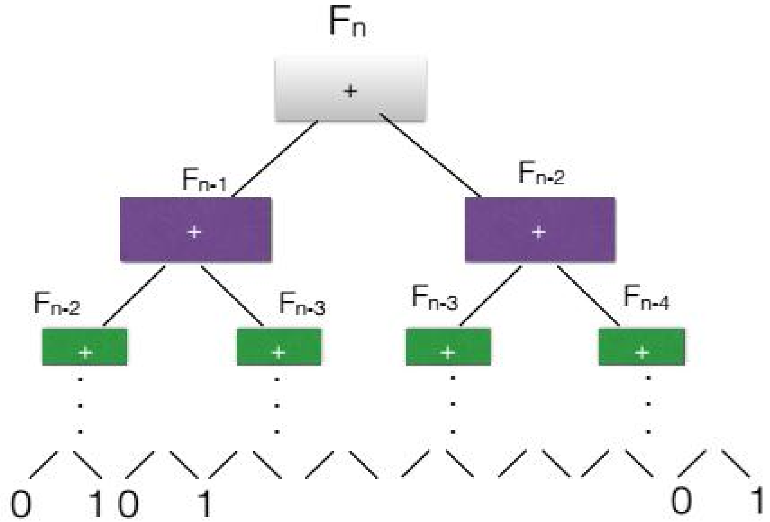
```
RecFibo(n):
  if (n < 2)
      return n
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

Running time? (backtracking recurrence)

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

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

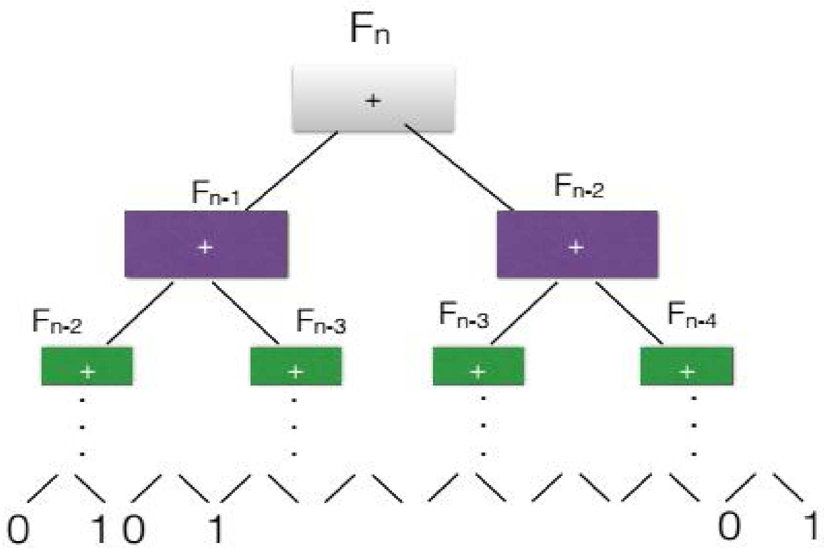
Running time via Rec Tree



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

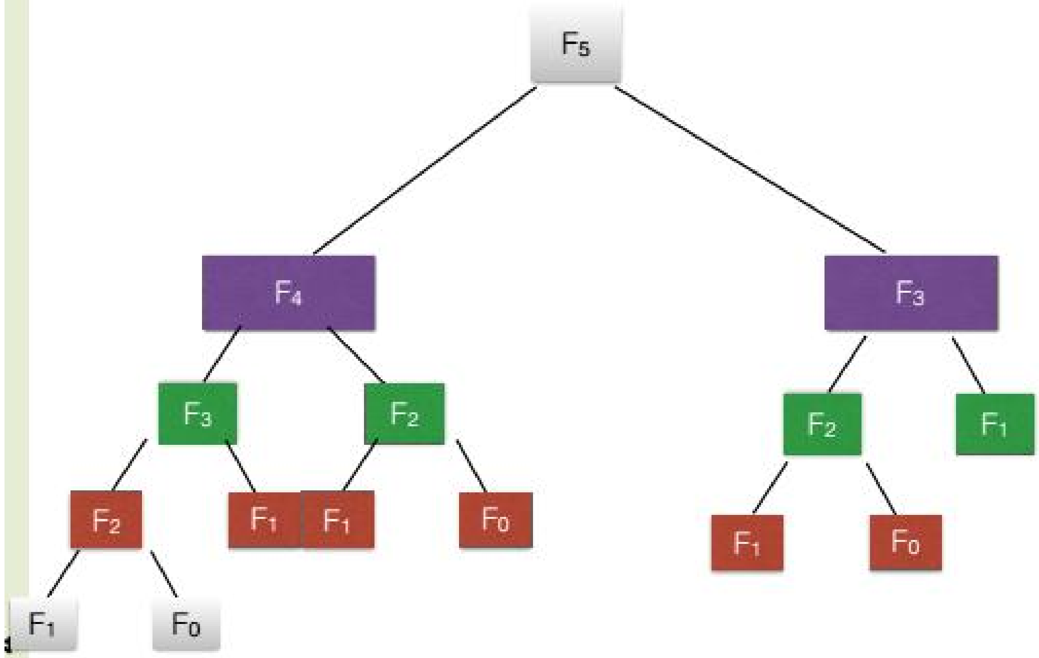
There are F_n 1s and F_{n-1} 0s
F_{n+1} leaves total!

Running time via Rec Tree

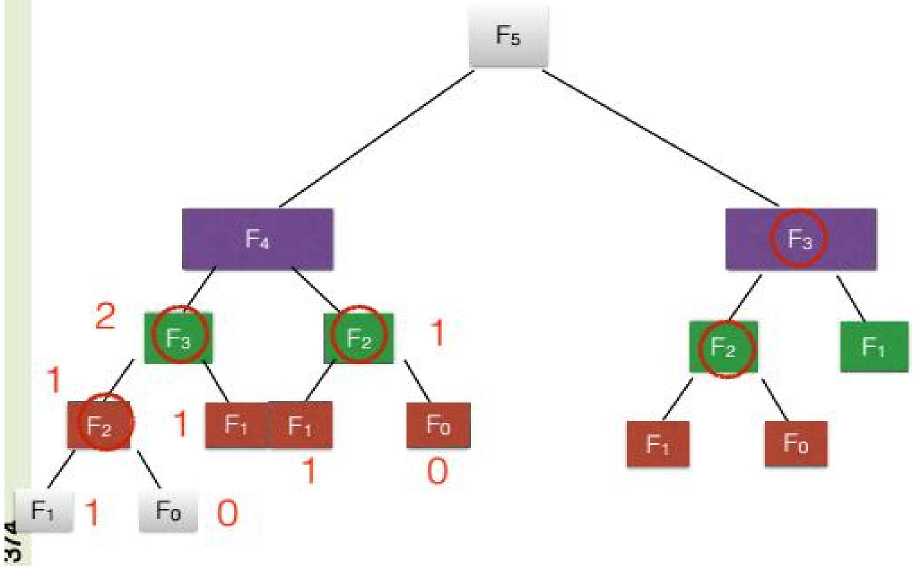


$2F_{n+1}$ -1 nodes (additions)

Running time via Rec Tree

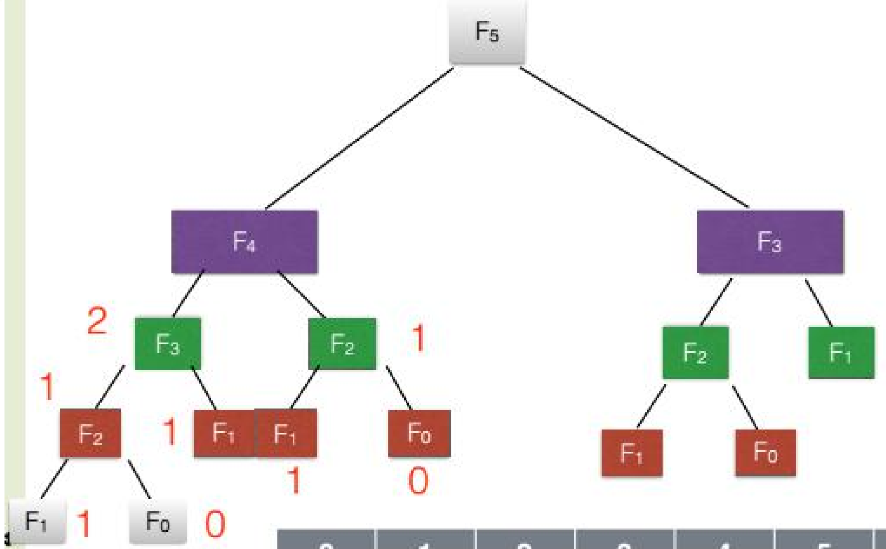


Running time via Rec Tree



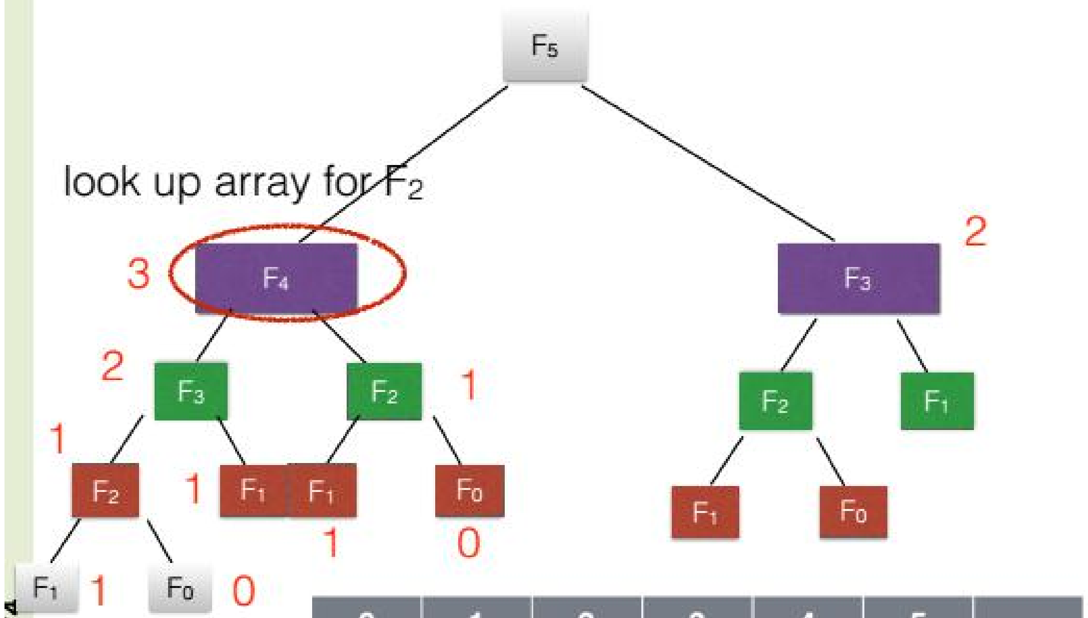
Keep an array to remember the previous values!

Running time via Rec Tree



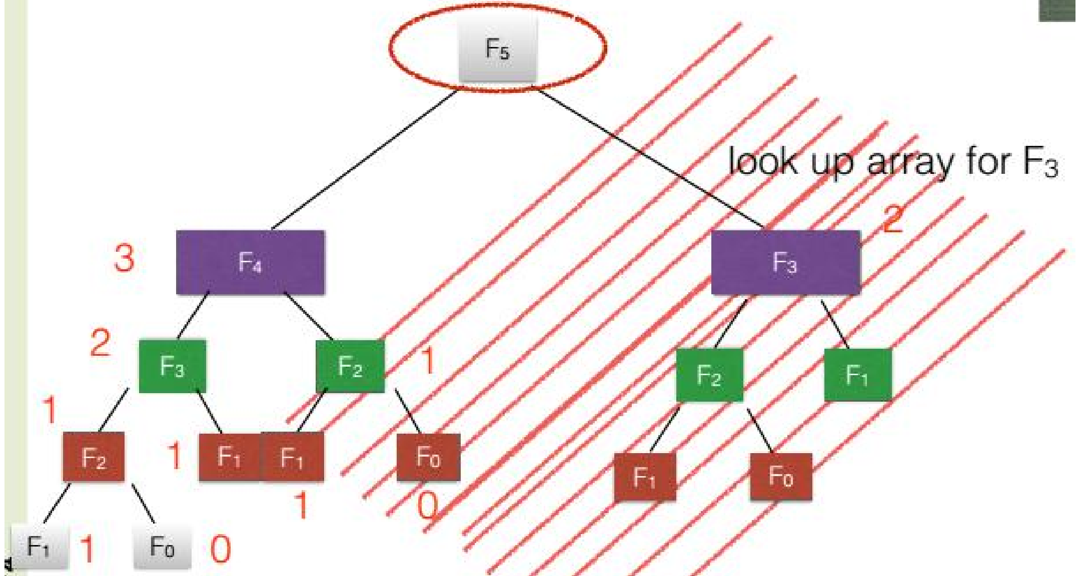
	0	1	2	3	4	5	...
F	0	1	1	2			

Running time via Rec Tree



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

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] ← 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] ← 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

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← 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] ← 0
  F[1] ← 1
  for i ← 2 to n        order
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

- 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

- 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?
- Saves space, sometimes important

Longest Increasing Subsequence (LIS)

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

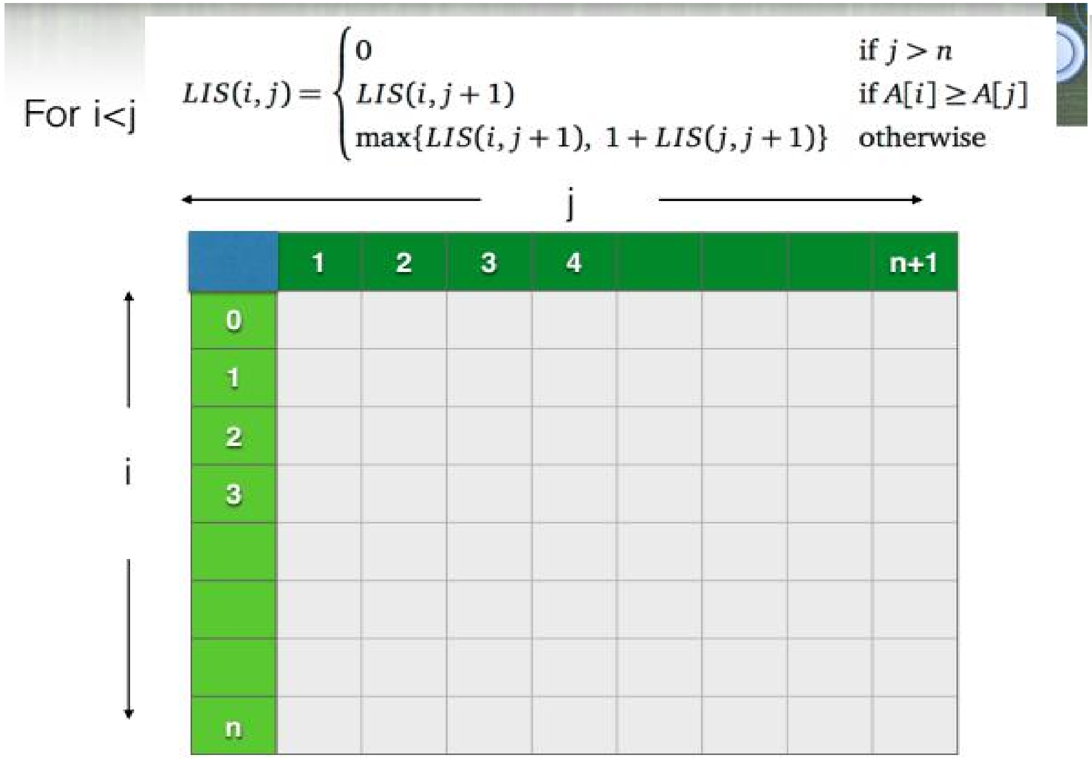
- LIS(A[1…n],p)=
  - 0 if n=0
  - LIS(A[2…n],p) if A[1]≤p
  - MAX { LIS(A[2…n],p)
    1+LIS(A[2…n],A[1])}

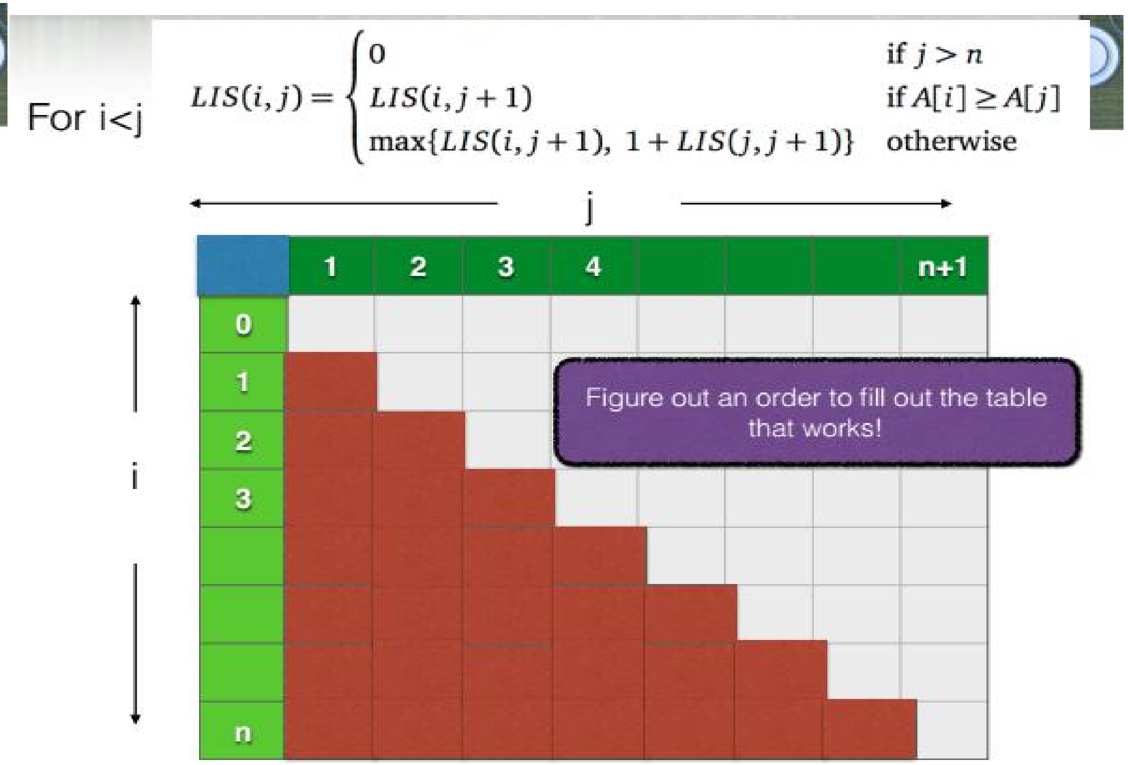
Longest Increasing Subsequence (LIS)

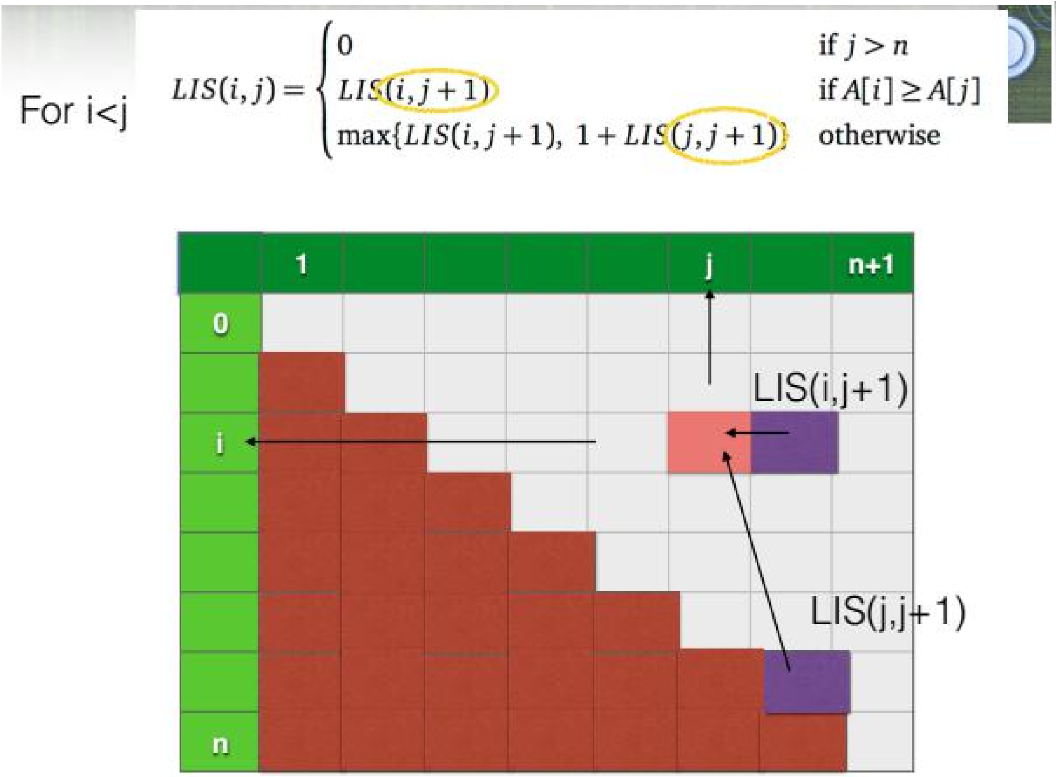
For i<j

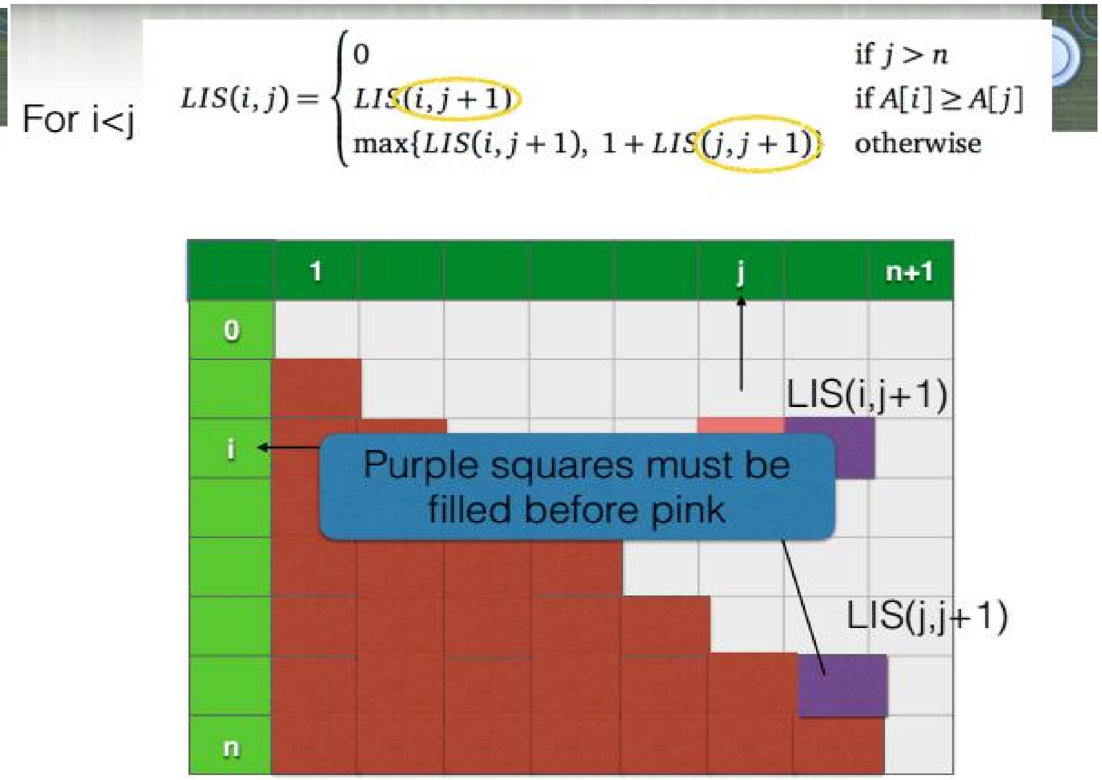
$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$

- 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
$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$
j
i
1
2
3
4
n+1
0
1
2
3
n

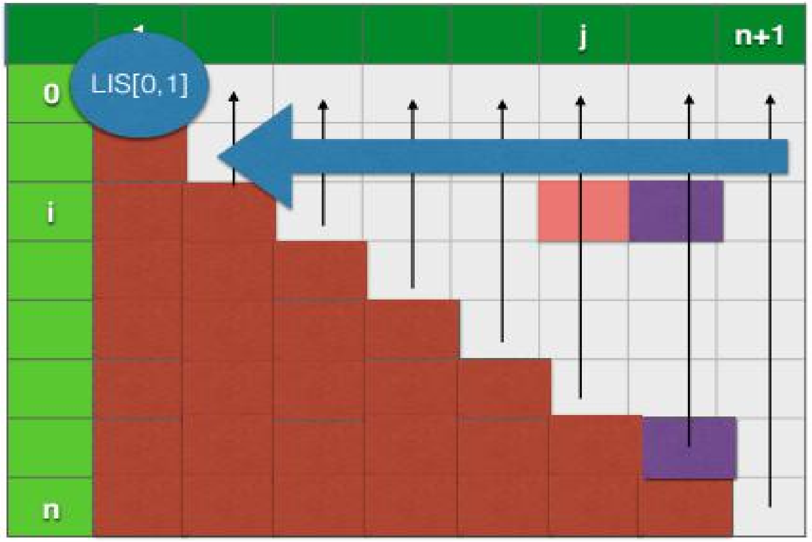

For i<j
$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$
j
i
1
2
3
4
n+1
0
1
2
3
n
Figure out an order to fill out the table that works!


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

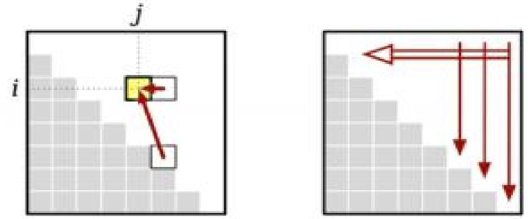

For i<j
$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$
1
j
n+1
0
i
n
LIS(i,j+1)
Purple squares must be filled before pink
LIS(j,j+1)

For i<j

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



Longest Increasing Subsequence (LIS)



doesn'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 man 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]
```


Exercise

Definition

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

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

Example

MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM has **MHYMRORMYHM** as a palindromic subsequence