Pre-lecture brain teaser

We saw a linear time selection algorithm in the previous lecture.

Why did we choose lists of size 5? Will lists of size 3 work?

(Hint) Write a recurrence to analyze the algorithm's running time if we choose a list of size k.

ECE-374-B: Lecture 11 - Backtracking and memorization

Instructor: Nickvash Kani

University of Illinois at Urbana-Champaign

Pre-lecture brain teaser

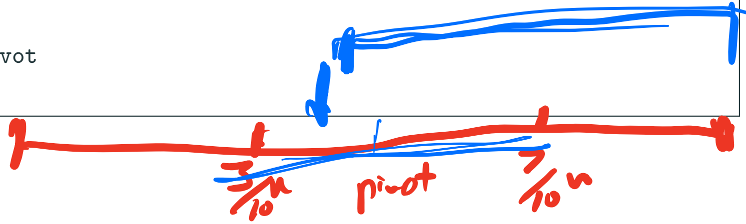
We saw a linear time selection algorithm in the previous lecture.

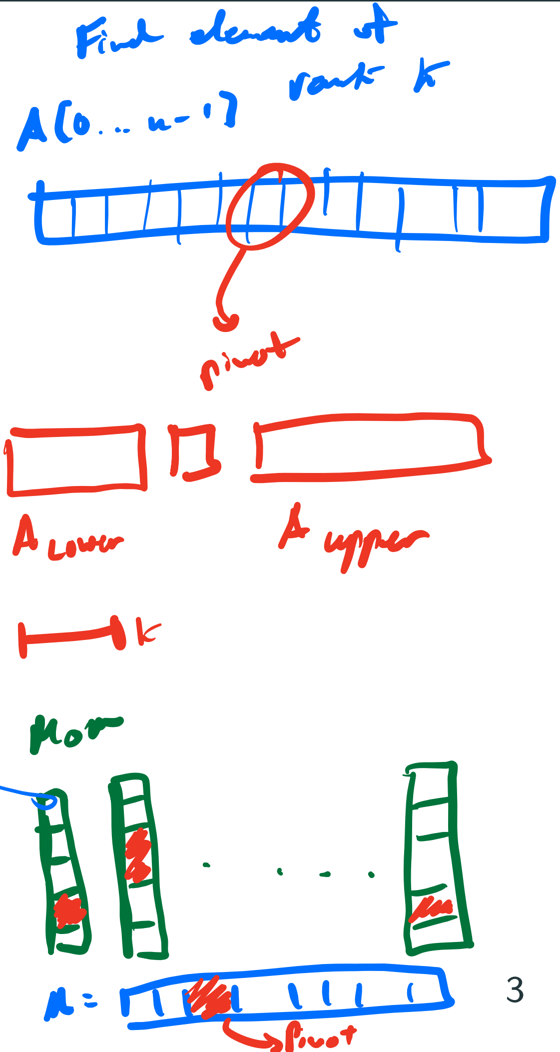
Why did we choose lists of size 5? Will lists of size 3 work?

(Hint) Write a recurrence to analyze the algorithm's running time if we choose a list of size k.

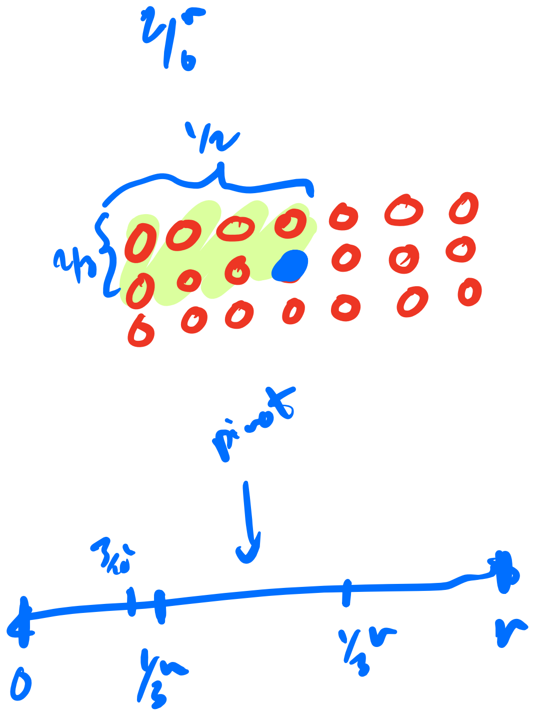
Median of medians time analysis

```
Median-of-medians(A, i):
    sublists = [A[j:j+5] for j ∈range(0, len(A), 5)]
    medians = [sorted (sublist)[len (sublist)/2] for sublist ∈sublists]

    // Base Case
    if len (A) ≤ 5 return sorted (a)[i]

    // Find median of medians
    if len (medians) ≤ 5
        pivot = sorted (medians)[len (medians)/2]
    else
        pivot = Median-of-medians (medians, len/2)

    // Partitioning Step
    low = [j for j ∈A if j < pivot]
    high = [j for j ∈A if j > pivot]

    k = len (low)
    if i < k
        return Median-of-medians (low, i)
    elseif i > k
        return Median-of-medians (low, i-k-1)
    else
    return pivot
```





Median of medians time analysis



```
Median-of-medians(A, i):
    sublists = [A[j:j+5] for j ∈range(0, len(A), 5)]
    medians = [sorted (sublist)[len (sublist)/2] for sublist ∈sublists]

    // Base Case
    if len (A) ≤5 return sorted (a)[i]

    // Find median of medians
    if len (medians) ≤5
        pivot = sorted (medians)[len (medians)/2]
    else
        pivot = Median-of-medians (medians, len/2)

    // Partitioning Step
    low = [j for j ∈A if j < pivot]
    high = [j for j ∈A if j > pivot]

    k = len (low)
    if i < k
        return Median-of-medians (low, i)
    elseif i > k
        return Median-of-medians (low, i-k-1)
    else
    return pivot
```

$$T(n) = T(\frac{1}{5}n) + T(\frac{7}{10}n) + cn$$

Brain teaser

We saw a linear time selection algorithm in the previous lecture.

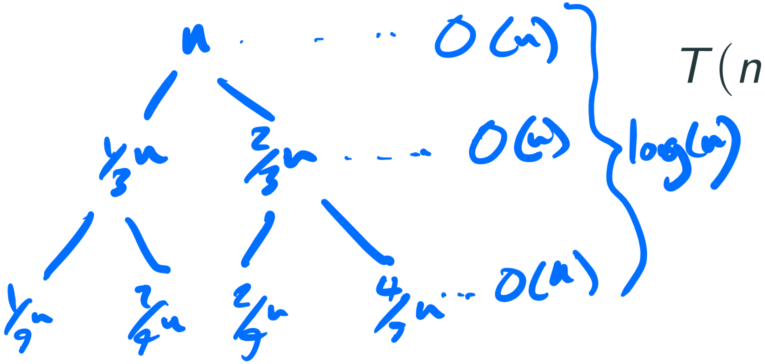
Why did we choose lists of size 5? Will lists of size 3 work?

$T(n) =$

Brain teaser

We saw a linear time selection algorithm in the previous lecture.

Why did we choose lists of size 5? Will lists of size 3 work?



$$T(n) = T(\frac{1}{3}n) + T(\frac{4}{6}n) + cn$$

$= O(n \log n)$

Brain teaser

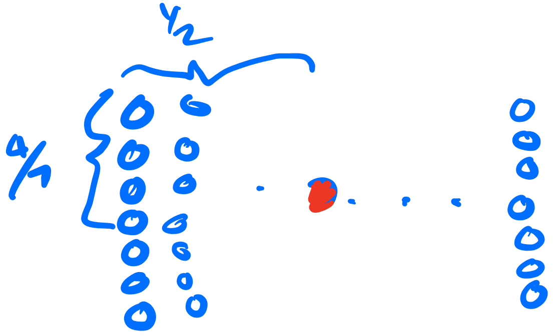
We saw a linear time selection algorithm in the previous lecture.

Why did we choose lists of size 5? Will lists of size 3 work?

$$T(n) = T(\frac{1}{3}n) + T(\frac{4}{6}n) + cn$$

What about $k = 7$?

$$T(n) = T(\frac{1}{7}n) + T(\frac{10}{14}n) + cn$$



Brain teaser

We saw a linear time selection algorithm in the previous lecture.

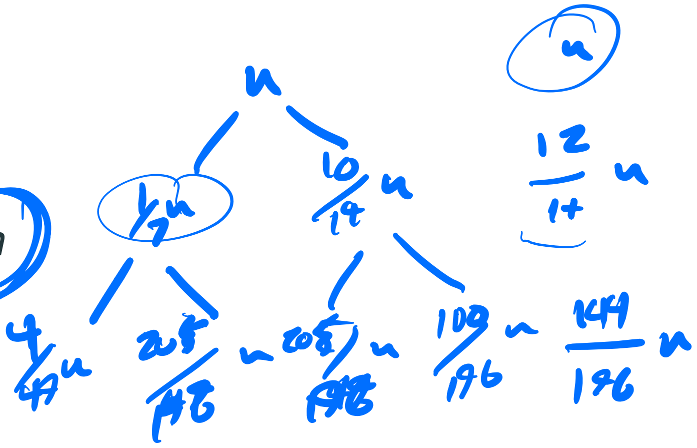
Why did we choose lists of size 5? Will lists of size 3 work?

$$T(n) = T(\frac{1}{3}n) + T(\frac{4}{6}n) + cn$$

What about $k = 7$?

$$T(n) = T(\frac{1}{7}n) + T(\frac{10}{14}n) + cn$$

$= O(n)$



On different techniques for recursive algorithms

Recursion

Reduction: Reduce one problem to another

Recursion
A special case of reduction

- reduce problem to a smaller instance of itself
- self-reduction

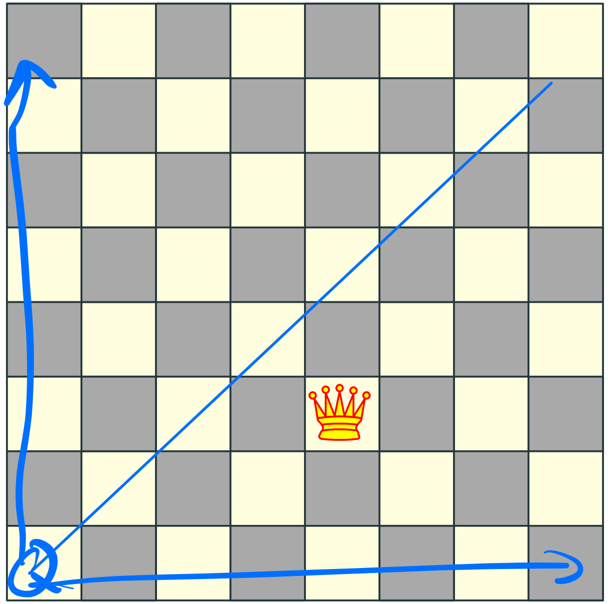
- Problem instance of size n is reduced to one or more instances of size $n-1$ or less.
- For termination, problem instances of small size are solved by some other method as base cases.

Recursion in Algorithm Design

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

Search trees and backtracking

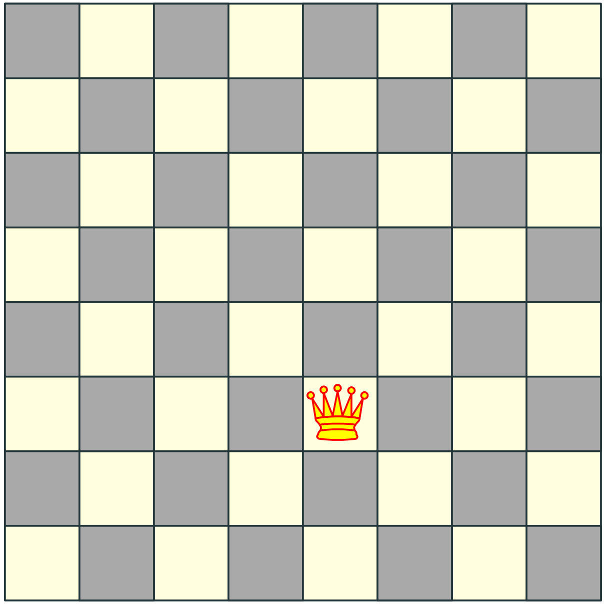
The queens problem



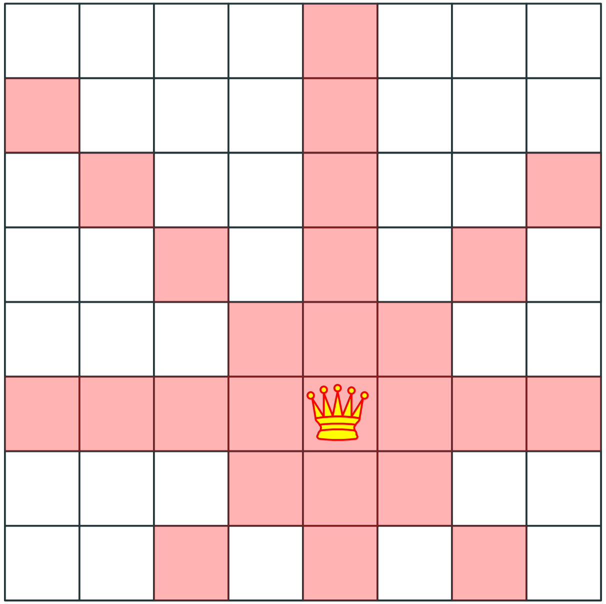
Q: How many queens can one place on the board?

Q: Can one place 8 queens on the board?

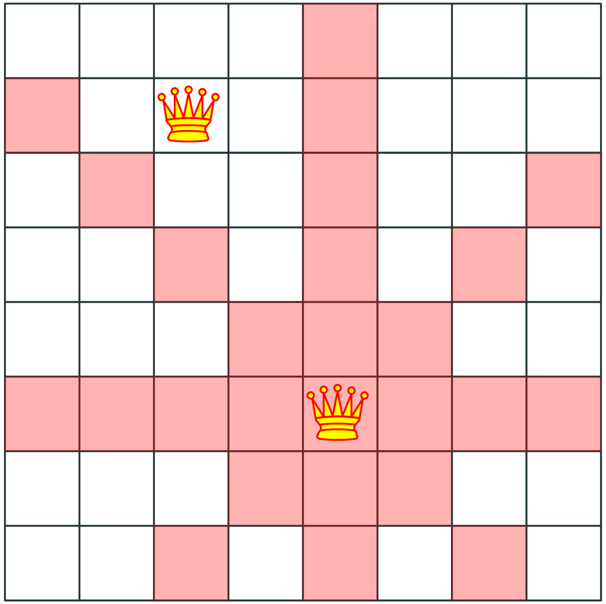
The queens problem



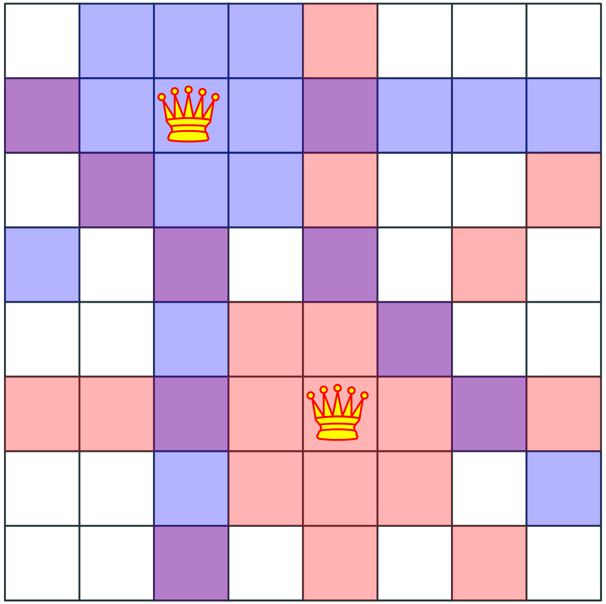
The queens problem



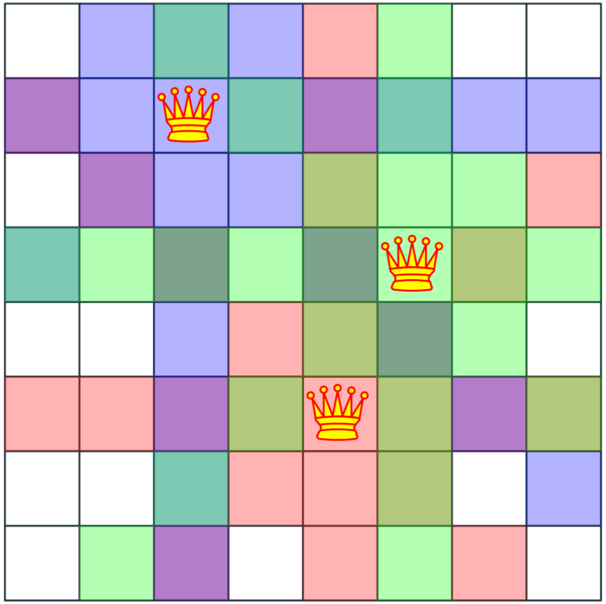
The queens problem



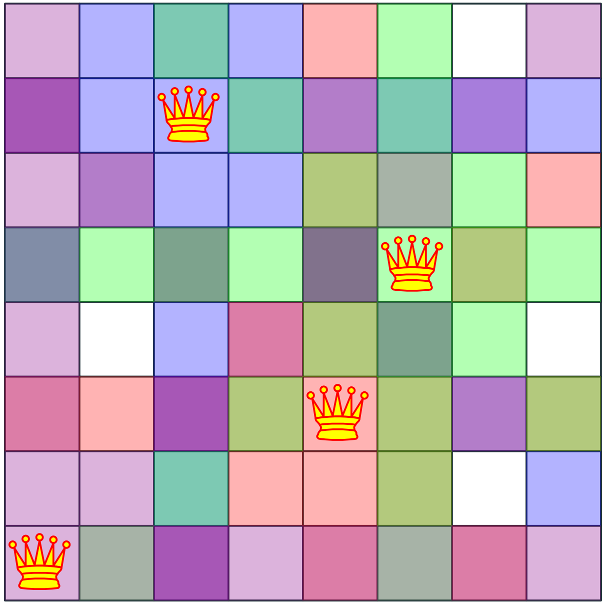
The queens problem



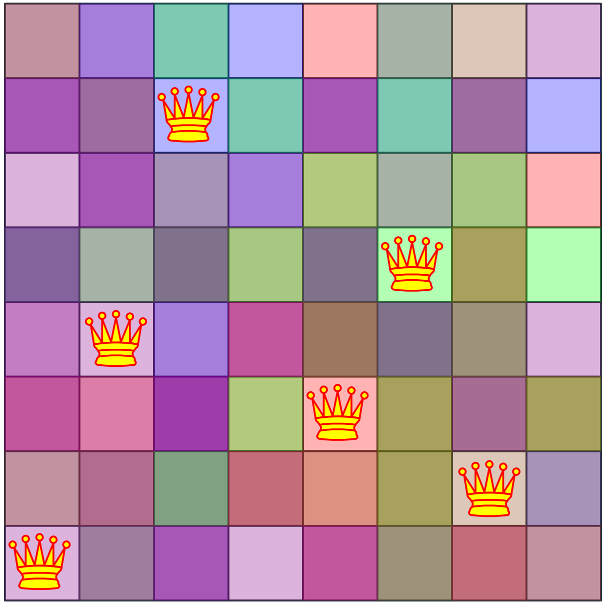
The queens problem



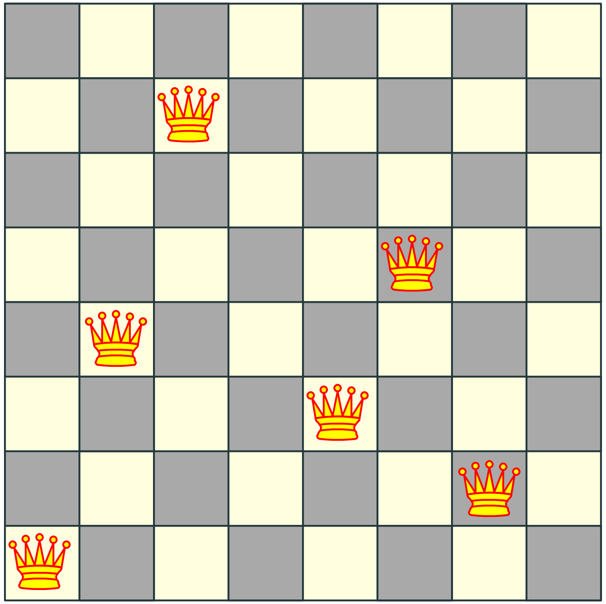
The queens problem



The queens problem



The queens problem



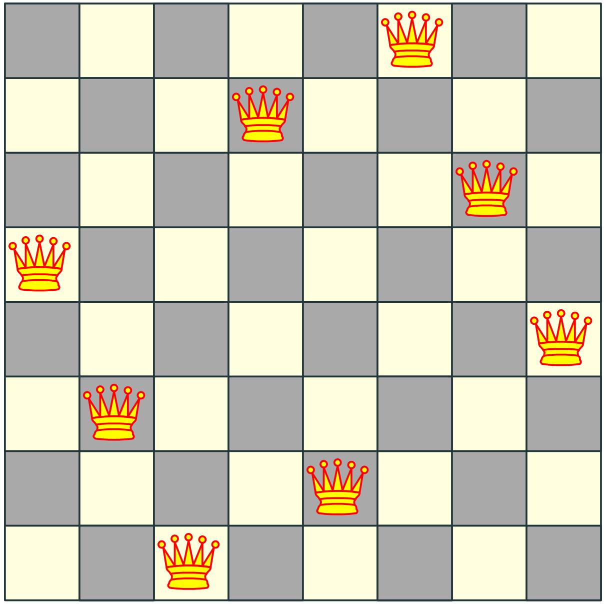
Q: How many queens can one place on the board? 6

Q: Can one place 8 queens on the board? How many permutations?

No

The eight queens puzzle

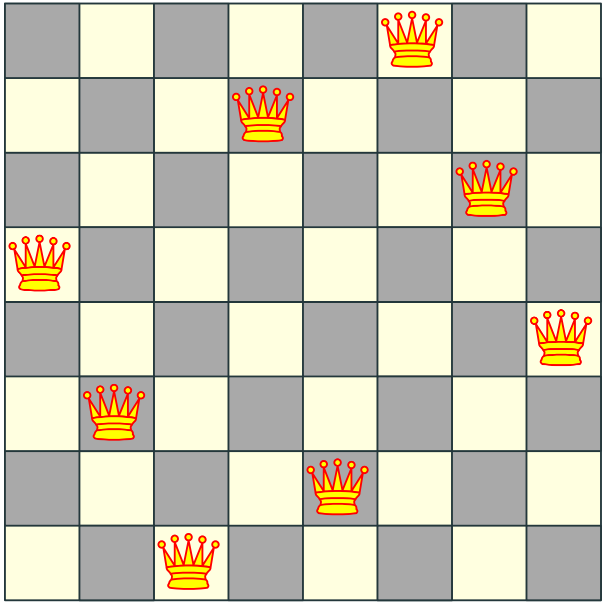
Problem published in 1848, solved in 1850.



96 solutions

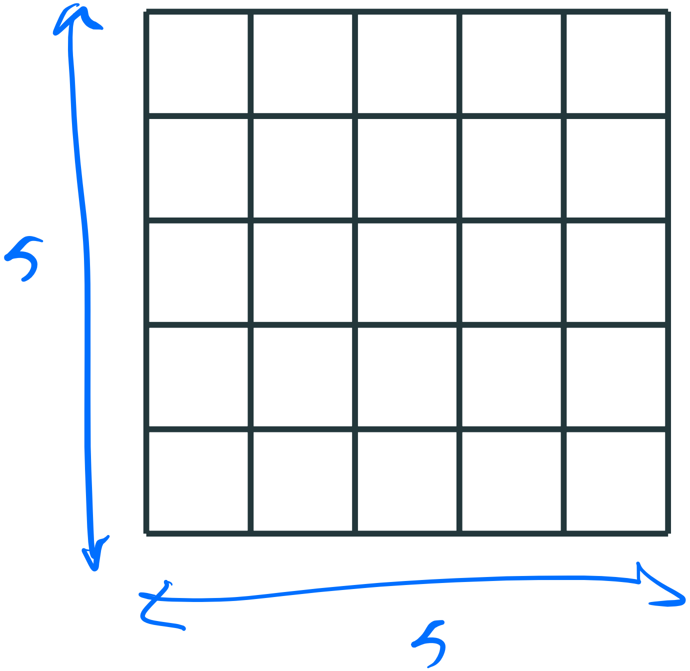
The eight queens puzzle

Problem published in 1848, solved in 1850.



Q: How to solve problem for general n?

Introducing concept of state tree

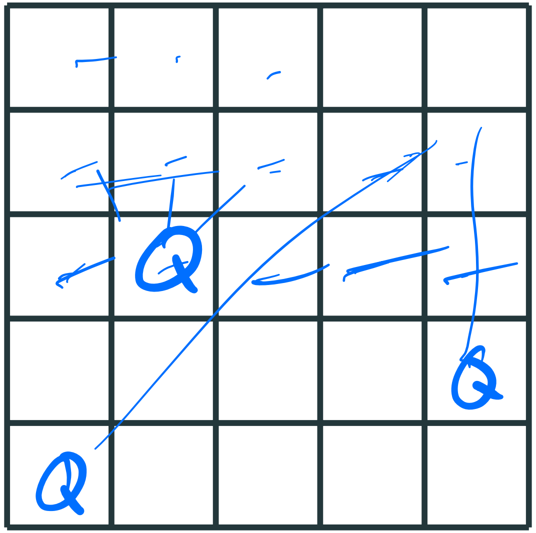


What if we attempt to find all the possible permutations and then check?

Search tree for 5 queens

$^{n^2}C_n$ $\quad {}^{n}C_r$ $\quad \frac{n!}{(n-r)!\,r!}$ $\quad O(n^n)$

$\frac{n^2!}{(n^2-n)!\,n!}$



Brute Force:

- Check all possible permutations of n queens on the board
- Check if queens are guarding each other,
  - if not
    - return permutation
  - if so
    - new permutation

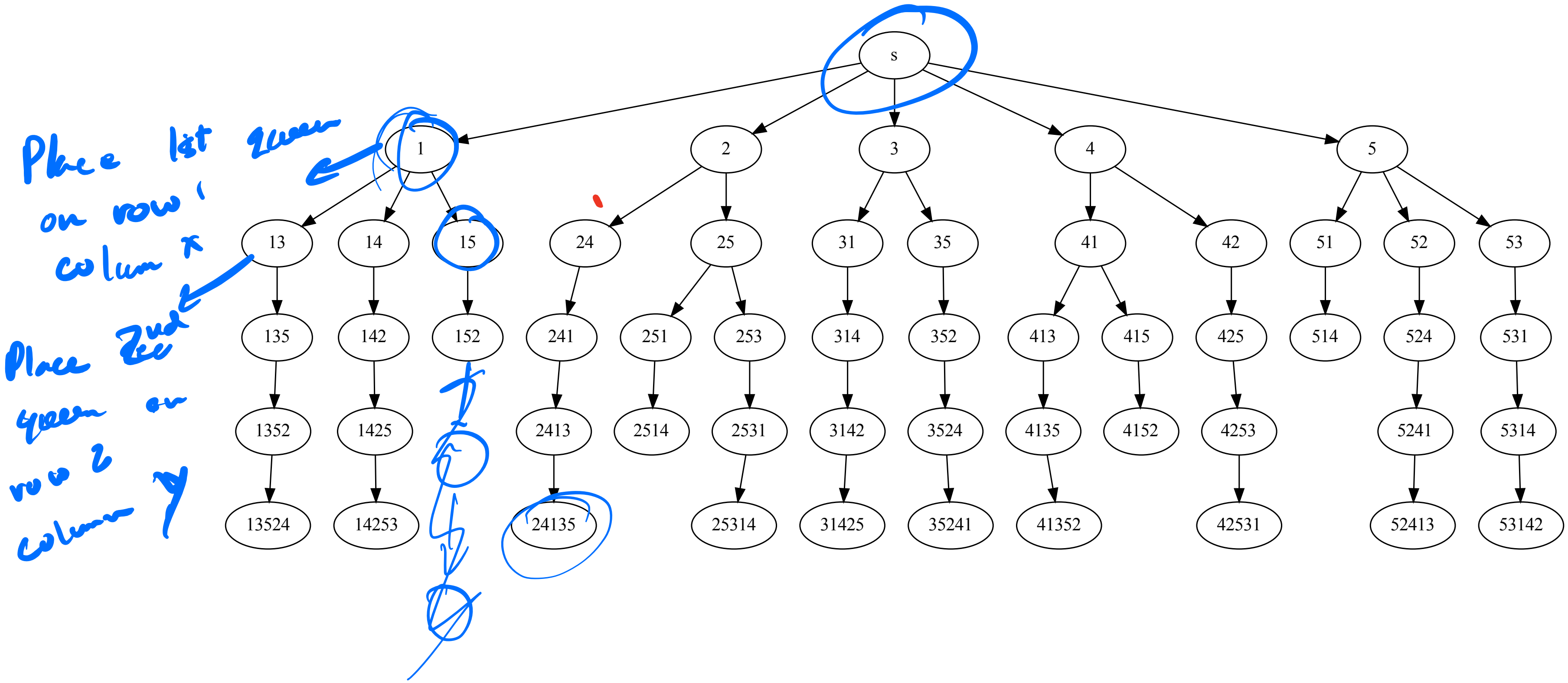
Let's be a bit smarter and recognize that:

- Queens can't be on the same row, column or diagonal
- Can have n queens max.

$(n^2)^n = O(n^{2n})$

$O(n^2 n^{2n}) \ldots$

Search tree for 5 queens



Backtracking: Informal definition

Recursive search over an implicit tree, where we “backtrack” if certain possibilities do not work.

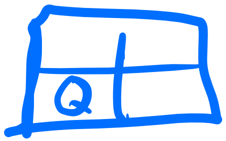
n queens C++ code

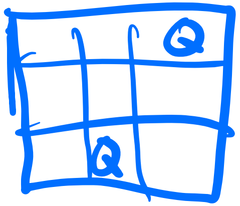
```
void  generate_permutations( int  * permut, int  row, int  n )
{
   if  ( row == n ) {
      print_board( permut, n );
      return;
   }

   for  ( int  val = 1; val <= n; val++ )
      if  ( isValid( permut, row, val ) ) {
         permut[ row ] = val;
         generate_permutations( permut, row + 1, n );
      }
}

generate_permutations( permut, 0, 8 );
```

$O(n^n) > O(n!)$

Quick note: n queens - number of solutions

N	Number of Solutions	Number of Unique Solutions
1	1	1
2	0	0
3	0	0
4	2	1
5	10	2
6	4	1
7	40	6
8	92	12
9	352	46
10	724	92
11	2,680	341
12	14,200	1,787
13	73,712	9,233
14	365,596	45,752
15	2,279,184	285,053





Sudoku

Sudoku problem

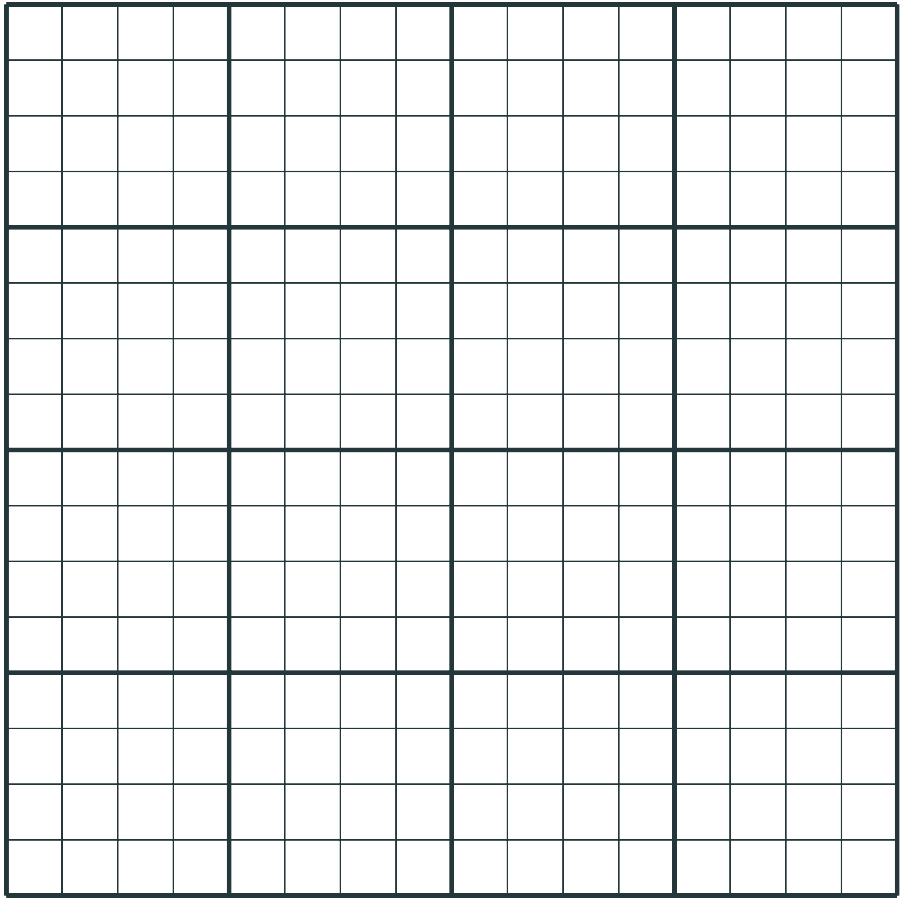
	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

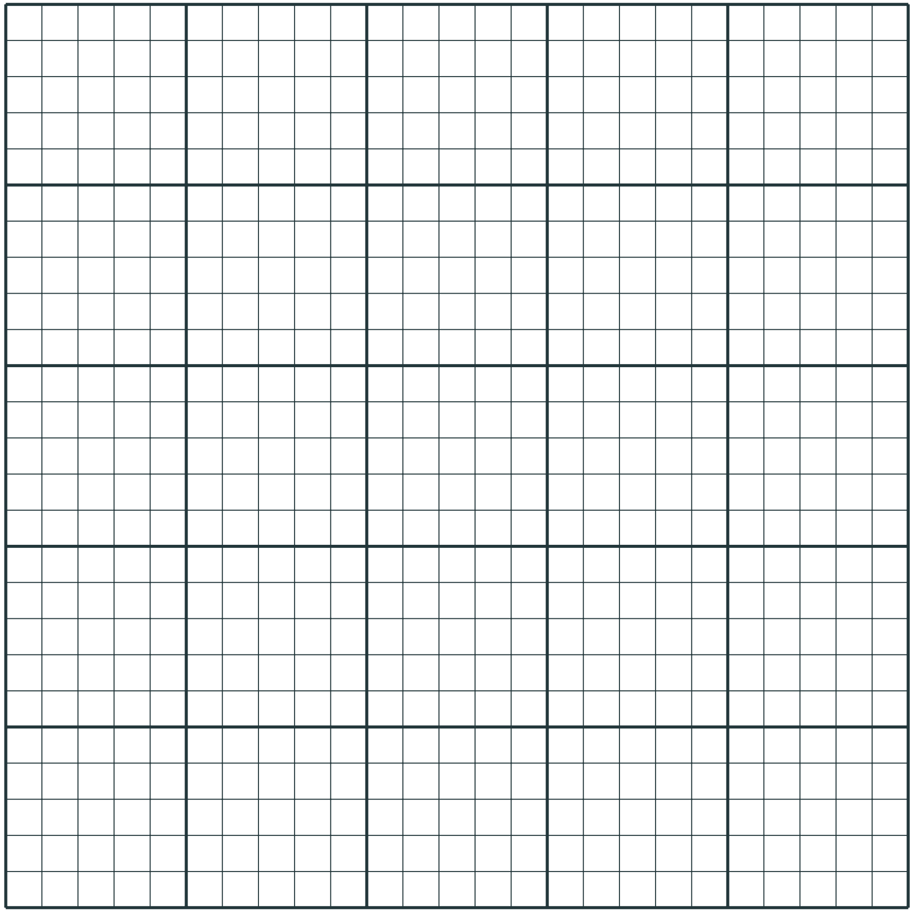
Unsolved Sudoku

4	2	*6*	5	*7*	1	*3*	9	*8*
8	*5*	*7*	2	*9*	3	*1*	*4*	6
1	3	*9*	*4*	6	*8*	*2*	7	*5*
9	*7*	1	*3*	*8*	*5*	6	*2*	*4*
5	4	*3*	*7*	*2*	*6*	*8*	1	9
6	*8*	2	*1*	*4*	*9*	7	*5*	*3*
7	9	*4*	*6*	3	*2*	*5*	8	*1*
2	*6*	*5*	8	*1*	4	*9*	*3*	7
3	1	*8*	9	*5*	7	*4*	6	*2*

Solved Sudoku

Variable Sized Sudoku





Naive Enumeration

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

```
algSudokuNaive(S[0..n − 1, 0..n − 1]):
    for possible value (X) in empty space do
        if SudokuValid? == True then
            return X

    return NULL
```

Naive Enumeration

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

algSudokuNaive$(S[0..n-1, 0..n-1])$:

```
    for possible value (X) in empty space do
        if SudokuValid? == True then
            return X

    return NULL
```

Running time:

Naive Enumeration

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

```
algSudokuNaive(S[0..n-1, 0..n-1]):
    for possible value (X) in empty space do
        if SudokuValid? == True then
            return X

    return NULL
```

Running time: $O(n^2 9^{n^2})$.

n^2 time to check all rows/columns/squares contain values 1 through n

9 possibilities per square for n^2 squares

Better Enumeration

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

```
Initialize Bitmap (BM) to contain only
    values available for each square
algSudoku-smaller(S[0..n − 1, 0..n − 1], BM[0..n − 1, 0..n − 1]):
    for each empty space X do
        for each possible value x for X according to BM do
            S-new = S(Assign X = x)
            BM-new = Modify BM removing x from same
                        row/column/square
            if no more empty squares
                return X
            else
                algSudoku-smaller(S, BM)

    return NULL
```

Better Enumeration

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

```
Initialize Bitmap (BM) to contain only
    values available for each square
algSudoku-smaller(S[0..n−1, 0..n−1], BM[0..n−1, 0..n−1]):
    for each empty space X do
        for each possible value x for X according to BM do
            S-new = S(Assign X = x)
            BM-new = Modify BM removing x from same
                        row/column/square
            if no more empty squares
                return X
            else
                algSudoku-smaller(S, BM)

    return NULL
```

Running time:

Better Enumeration

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

```
Initialize Bitmap (BM) to contain only
    values available for each square
algSudoku-smaller(S[0..n-1, 0..n-1], BM[0..n-1, 0..n-1]):
    for each empty space X do
        for each possible value x for X according to BM do
            S-new = S(Assign X = x)
            BM-new = Modify BM removing x from same
                        row/column/square
            if no more empty squares
                return X
            else
                algSudoku-smaller(S, BM)

    return NULL
```

Running time: $O(9^{n^2})$.

9 possibilities per square for n^2 squares

Longest Increasing Sub-sequence

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

- Sequence: $6, 3, 5, 2, 7, 8, 1, 9$
- 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: $2, 7, 9$.

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

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

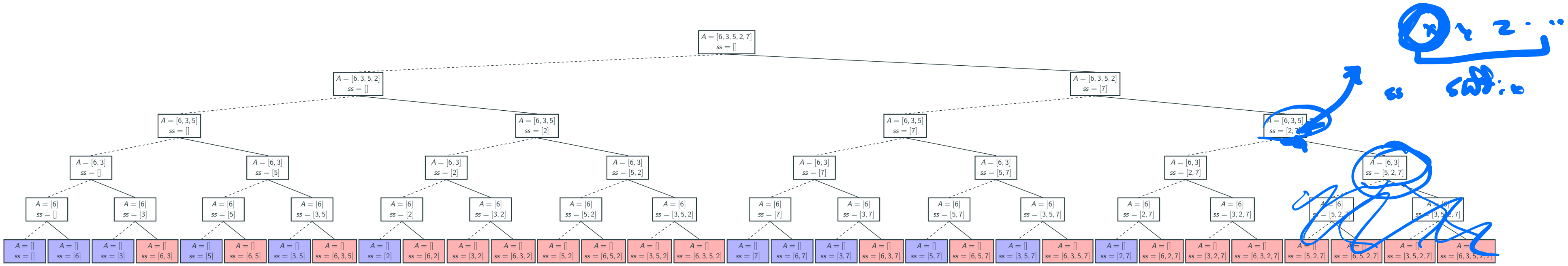
- Sequence: 6, 3, 5, 2, 7, 8, 1
- Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
- Longest increasing subsequence: 3, 5, 7, 8

Naive Enumeration

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

```
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
```

Naive Recursion Enumeration - State Tree



- This is just for [6,3,5,2,7]! (Tikz won't print larger trees)
- How many leafs are there for the full [6,3,5,2,7, 8, 1] sequence 2^n
- What is the running time? $O(n 2^n)$

Naive Enumeration

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

```
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:

Naive Enumeration

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

```
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.

Recursive Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

Recursive Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS$(A[1..n])$:

- Case 1: Does not contain $A[n]$ in which case LIS$(A[1..n])$ = LIS$(A[1..(n-1)])$
- Case 2: contains $A[n]$ in which case LIS$(A[1..n])$ is

Recursive Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS$(A[1..n])$:

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

Recursive Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

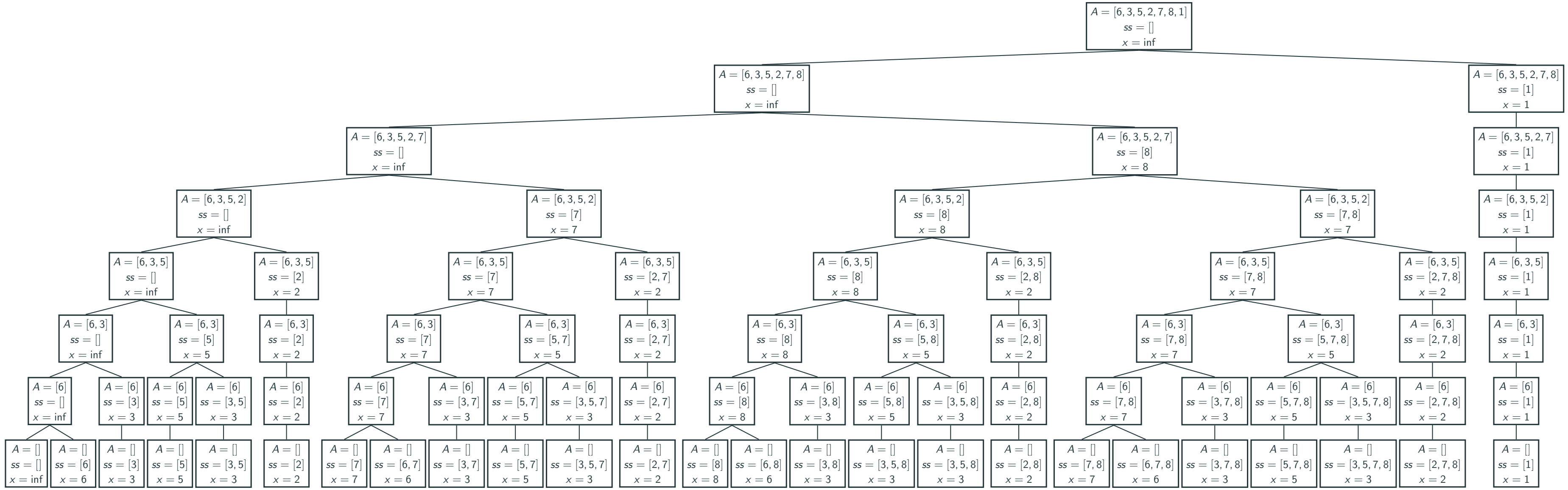
- Case 1: Does not contain $A[n]$ in which case LIS($A[1..n]$) = LIS($A[1..(n-1)]$)
- Case 2: contains $A[n]$ in which case LIS($A[1..n]$) is not so clear.

Observation

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

Example

Sequence: $A[0..6] = 6, 3, 5, 2, 7, 8, 1$



Recursive Approach

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**
    $m = max(m, 1 +$ **LIS_smaller**$(A[1..(n-1)], A[n]))$
  `Output` m

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

Running time analysis

Running time of LIS([1..n])

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$(A[1..n], \infty)$

$A = [1, 2, 3, 4, \ldots n]$

Running time of LIS([1..n])

Lemma
LIS_smaller *runs in* $O(2^n)$ *time.*

Running time of LIS([1..n])

Lemma
LIS_smaller *runs in $O(2^n)$ time.*

Improvement: From $O(n2^n)$ to $O(2^n)$.

Running time of LIS([1..n])

Lemma
LIS_smaller *runs in* $O(2^n)$ *time.*

Improvement: From $O(n2^n)$ to $O(2^n)$.

....one can do much better using memorization!

$\underbrace{n \cdot n \cdot n \cdot \ldots \cdot n}_{n}$ $O(n^n)$