Pre-lecture brain teaser

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

Quick Select + MoM

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 memoization

Instructor: Abhishek Kumar Umrawal

February 27, 2024

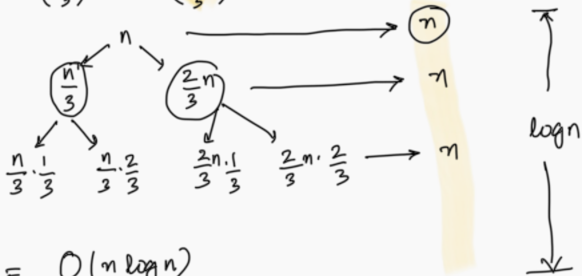
University of Illinois at Urbana-Champaign

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*.

$k=3$: $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2}{3}n\right) + O(n)$



$\Rightarrow T(n) = O(n \log n)$

$k=7:$ $\quad T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5}{7}n\right) + O(n)$

Learning Objectives

Learning Objectives

At the end of the lecture, you should be able to understand

- the details of the quickselect and medians of median algorithms,
- the idea of backtracking through the 8-queens puzzle,
- the longest increasing subsequence problem and recursive algorithms to solve it,
- the intuition behind memoization.

Given an array $A = [0, \ldots, n-1]$ of n numbers and an index i, where $0 \le i \le n-1$, find the i^{th} smallest element of A.

For instance, assume $n = 20$ and $i = 10$.

4	3	15	7	1	17	9	10	14	13	8	18	11	2	12	16	6	19	5	20

The smallest element of rank 10 would be 11. But how do we figure that out

Do median of medians.....

(MoM)

Call Median-of-Medians(A, 10)

Quick select + MoM

Review linear time selection

Given an array $A = [0, ..., n-1]$ of n numbers and an index i, where $0 \leq i \leq n-1$, find the i^{th} smallest element of A.

For instance, assume $n = 20$ and $i = 10$.

4	3	15	7	1	17	9	10	14	13	8	18	11	2	12	16	6	19	5	20

The smallest element of rank 10 would be 11. But how do we figure that out

Do median of medians.....

Call Median-of-Medians(A, 10)

First thing we need to do is find the pivot!

Review linear time selection

Given an array $A = [0, ..., n-1]$ of n numbers and an index i, where $0 \leq i \leq n-1$, find the i^{th} smallest element of A.

For instance, assume $n = 20$ and $i = 10$.

4	3	15	7	1	17	9	10	14	13	8	18	11	2	12	16	6	19	5	20

The smallest element of rank 10 would be 11. But how do we figure that out

Do median of medians.....

Call Median-of-Medians(A, 10)

First thing we need to do is find the pivot!

Review linear time selection

First we reorganize:

4	17	8	16
3	9	18	6
15	10	11	19
7	14	2	5
1	13	12	20

Review linear time selection

First we reorganize:

$4 = n/5$

5

4	17	8	16
3	9	18	6
15	10	11	19
7	14	2	5
1	13	12	20

Then we sort each column:

1	9	2	5
3	10	8	6
4	13	11	16
7	14	12	19
15	17	18	20

Review linear time selection

First we reorganize: $\frac{n}{5} = \frac{20}{5}$

4

5

4	17	8	16
3	9	18	6
15	10	11	19
7	14	2	5
1	13	12	20

Then we sort each column:

1	9	2	5
3	10	8	6
4	13	11	16
7	14	12	19
15	17	18	20

4 #s $= \frac{n}{5}$

Still need the pivot. Find median of medians

Review linear time selection

1	9	2	5
3	10	8	6
4	13	11	16
7	14	12	19
15	17	18	20

Review linear time selection

1	9	2	5
3	10	8	6
4	13	11	16
7	14	12	19
15	17	18	20

$\leftarrow \frac{n}{5}$

4 11 13 16

- Call Median-of-Medians([4,13,11,16], floor(len/2) = 2)
- Can sort this in linear time.
- Get back 13.
- 13 is our new pivot!

Review linear time selection

Back to our original array! Use the pivot (=13) to break it up into two.

4	3	15	7	1	17	9	10	14	13	8	18	11	2	12	16	6	19	5	20

4	3	7	1	9	10	8	11	2	12	6	5

A_{lower}

13

15	17	14	18	16	19	20

A_{upper}

We know the following:

- $\text{len}(A_{Lower}) = 12$
- $\text{len}(A_{Upper}) = 7$
- Want $k = 10$

Review linear time selection

Back to our original array! Use the pivot (=13) to break it up into two.

4	3	15	7	1	17	9	10	14	13	8	18	11	2	12	16	6	19	5	20

4	3	7	1	9	10	8	11	2	12	6	5

13

15	17	14	18	16	19	20

We know the following:

- $\text{len}(A_{Lower}) = 12$
- $\text{len}(A_{Upper}) = 7$
- Want $k = 10$

Call Median-of-Medians$(A_{Lower}, 10)$

Review linear time selection

Then we do this again:

4	3	7	1	9	10	8	11	2	12	6	5

Review linear time selection

Then we do this again:

A_{Lower}:

4	3	7	1	9	10	8	11	2	12	6	5

First we reorganize:

4	10	
3	8	6
7	11	5
1	2	
9	12	

Then we do this again:

4	3	7	1	9	10	8	11	2	12	6	5

First we reorganize:

4	10	
3	8	6
7	11	5
1	2	
9	12	

Then we sort each column:

1	2	
3	8	5
4	10	6
7	11	
9	12	

Review linear time selection

1	2	
3	8	5
4	10	6
7	11	
9	12	

Review linear time selection

1	2	
3	8	5
4	10	6
7	11	
9	12	

- Call Median-of-Medians([4,10,6], floor(len/2) = 1) $\frac{n}{2}$ e.g. $\approx \frac{12}{5}$
- Can sort this in linear time.
- Get back 6.
- 6 is our new pivot!

length : $f(n)$

Review linear time selection

Back to our original array! Use the pivot (=12) to break it up into two (well three).

4	3	7	1	9	10	8	11	2	12	6	5

4	3	1	2	5

6

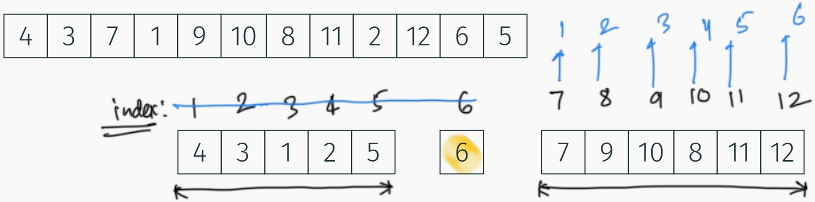
7	9	10	8	11	12

We know the following:

- $\text{len}(A_{Lower}) = 5$
- $\text{len}(A_{Upper}) = 6$
- Want $k = 10$ (pivot is of rank 6)

Review linear time selection

Back to our original array! Use the pivot (=12) to break it up into two (well three).



We know the following:

- $\text{len}(A_{Lower}) = 5$
- $\text{len}(A_{Upper}) = 6$
- Want $k = 10$ (pivot is of rank 6)

Call Median-of-Medians(A_{Upper}, $10 - 6 = 4$)

Review linear time selection

Then we do this again:

7	9	10	8	11	12

Then we do this again:

7	9	10	8	11	12

First we reorganize:

7	
9	
10	12
8	
11	

Review linear time selection

Then we do this again:

7	9	10	8	11	12

First we reorganize:

7	
9	
10	12
8	
11	

Then we sort each column:

7	
8	
9	12
10	
11	

Review linear time selection

7	
8	
9	12
10	
11	

Review linear time selection

7	
8	
9	12
10	
11	

- Call Median-of-Medians([9,12], floor(len/2) = 1)
- Can sort this in linear time.
- Get back 12.
- **12** is our new pivot!

Review linear time selection

Back to our original array! Use the pivot (=6) to break it up into two (well three).

7	9	10	8	11	12

7	9	10	8	11

12

We know the following:

- $\text{len}(A_{Lower}) = 5$
- $\text{len}(A_{Upper}) = 0$
- Want $k = 4$ (pivot is of rank 5)

Review linear time selection

Back to our original array! Use the pivot (=6) to break it up into two (well three).

7	9	10	8	11	12

7	9	10	8	11

12

We know the following:

- $\text{len}(A_{Lower}) = 5$
- $\text{len}(A_{Upper}) = 0$
- Want $k = 4$ (pivot is of rank 5)

Call Median-of-Medians(A_{Lower}, 4)

Final Step!

7	9	10	8	11

Can sort in linear time!

7	8	9	10	11

Return $Sorted(A[4]) = 11$

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
```

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

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?

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?

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

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?

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

What about $k = 7$?

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?

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

What about $k = 7$?

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

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

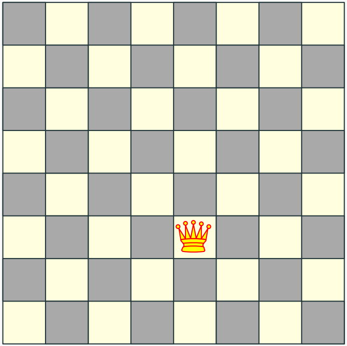
- Problem instance of size n is reduced to one or more instances of size $n-1$ or less.
- For termination, problem instances of small size are solved by some other method as base cases.

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 memoization to avoid recomputation of common solutions leading to iterative bottom-up algorithm.

Search trees and backtracking

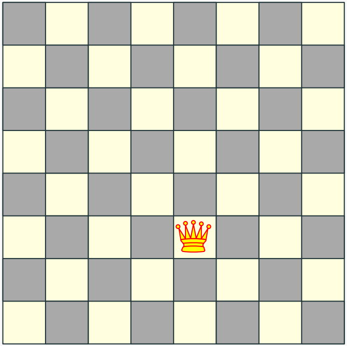
The queens problem



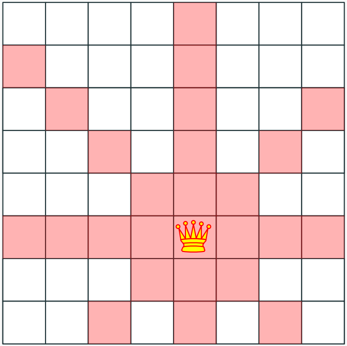
Q: How many queens can one place on the board?

Q: Can one place 8 queens on the board?

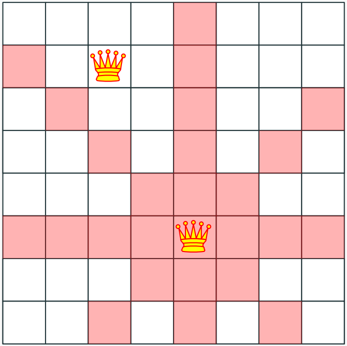
The queens problem



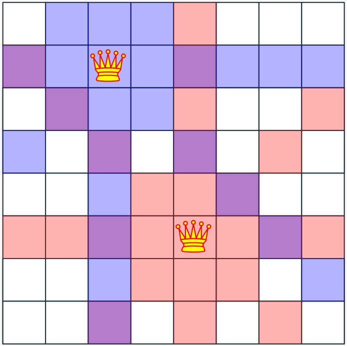
The queens problem



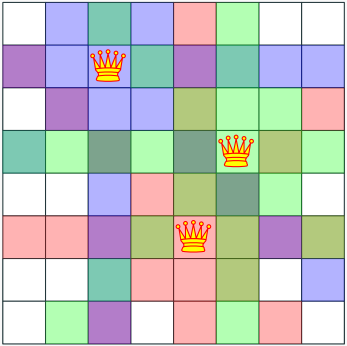
The queens problem



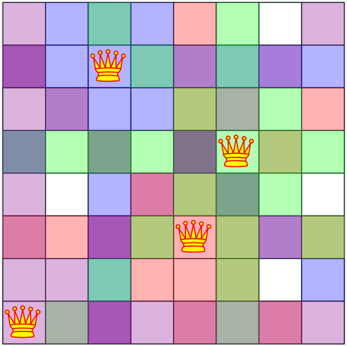
The queens problem



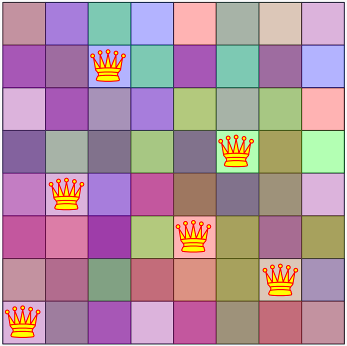
The queens problem

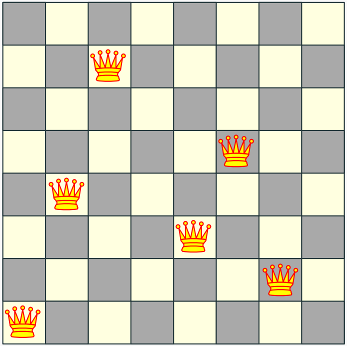


The queens problem



The queens problem





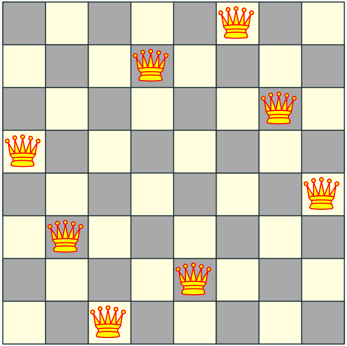
Q: How many queens can one place on the board?

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

The eight queens puzzle

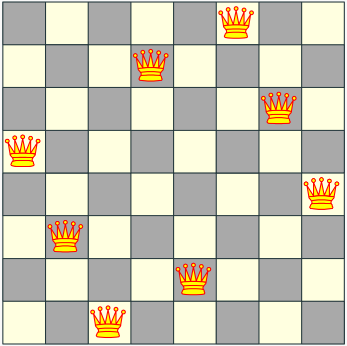
Max Bezzel Franz Nauck

Problem published in 1848, solved in 1850.



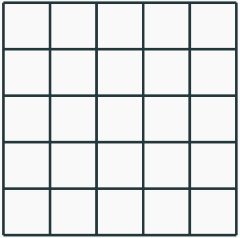
The eight queens puzzle

Problem published in 1848, solved in 1850.



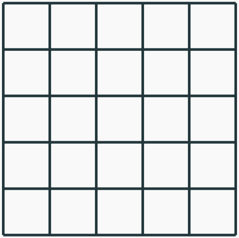
Q: How to solve problem for general n?

Introducing concept of state tree



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

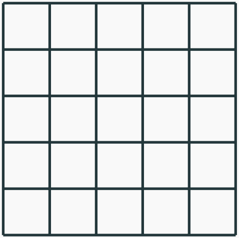
Search tree for 5 queens

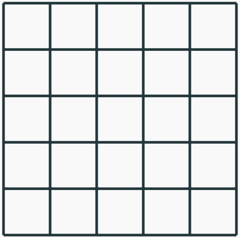


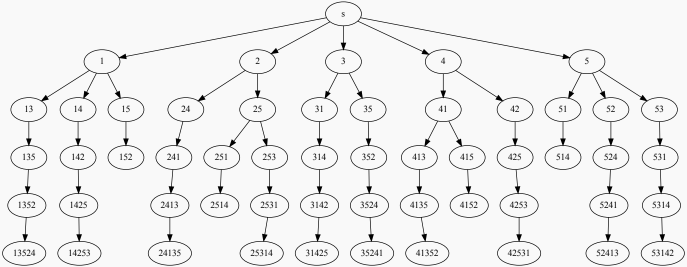
Let's be a bit smarter and recognize that:

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

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 );
```

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

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
$1 \le i_1 < i_2 < \ldots < i_k \le n$.

Definition
A sequence is increasing if $a_1 < a_2 < \ldots < a_n$. It is non-decreasing if $a_1 \le a_2 \le \ldots \le a_n$. Similarly decreasing and non-increasing.

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 <u>subsequence</u> 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 <u>increasing subsequence</u> $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ 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 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 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.

Recursive Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

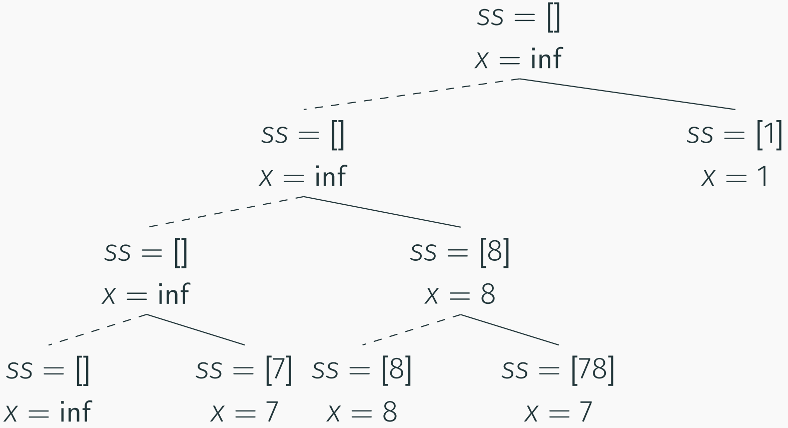
LIS($A[1..n]$):

- 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: $A[1..5] = 5, 9, 7, 8, 1$



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], ∞)
```

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], ∞)
```

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 memoization!