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 $k$. 
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 $k$. 

Assume $k$ is odd.
Review linear time selection

Given an array \( A = [0, \ldots, n - 1] \) 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 \( i = 10 \).

\[
\begin{array}{ccccccccccccccccccc}
4 & 3 & 15 & 7 & 1 & 17 & 9 & 10 & 14 & 13 & 8 & 18 & 11 & 2 & 12 & 16 & 6 & 19 & 5 & 20 \\
\end{array}
\]

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

Do median of medians.....

Call Median-of-Medians\((A, 10)\)
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 $i = 10$.

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 $A = [0, \ldots, n - 1]$ 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 $i = 10$.

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:

<table>
<thead>
<tr>
<th>4</th>
<th>17</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>
Review linear time selection

First we reorganize:

<table>
<thead>
<tr>
<th>4</th>
<th>17</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>

Then we sort each column:

<table>
<thead>
<tr>
<th>1</th>
<th>9</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>12</td>
<td>19</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>18</td>
<td>20</td>
</tr>
</tbody>
</table>

\[ n \cdot \frac{5}{8} \]
Review linear time selection

First we reorganize:

\[
\begin{array}{cccc}
4 & 17 & 8 & 16 \\
3 & 9 & 18 & 6 \\
15 & 10 & 11 & 19 \\
7 & 14 & 2 & 5 \\
1 & 13 & 12 & 20 \\
\end{array}
\]

Then we sort each column:

\[
\begin{array}{cccc}
1 & 9 & 2 & 5 \\
3 & 10 & 8 & 6 \\
4 & 13 & 11 & 16 \\
7 & 14 & 12 & 19 \\
15 & 17 & 18 & 20 \\
\end{array}
\]

Still need the pivot. Find median of medians
Review linear time selection

<table>
<thead>
<tr>
<th>1</th>
<th>9</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>12</td>
<td>19</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>18</td>
<td>20</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>9</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>11</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>12</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>18</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

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

We know the following:

- \( \text{len}(A_{\text{Lower}}) = 12 \)
- \( \text{len}(A_{\text{Upper}}) = 7 \)
- Want \( k = 10 \)
Review linear time selection

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

We know the following:

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

Call \textbf{Median-of-Medians}(A_{\text{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:

\[
\begin{array}{cccccccccccc}
4 & 3 & 7 & 1 & 9 & 10 & 8 & 11 & 2 & 12 & 6 & 5 \\
\end{array}
\]

First we reorganize:

\[
\begin{array}{ccc}
4 & 10 & \\
3 & 8 & 6 \\
7 & 11 & 5 \\
1 & 2 & \\
9 & 12 & \\
\end{array}
\]
Review linear time selection

Then we do this again:

First we reorganize:

Then we sort each column:

1 2
3 8 5
4 10 6
7 11
Review linear time selection

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

- Call Median-of-Medians ([4, 10, 6], floor(n/2) = 10)
- Can sort this in linear time.
- Get back 6.
- 6 is our new pivot!
Review linear time selection

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- Call **Median-of-Medians**([4,10,6], \(\text{floor}(n/2) = 10\))
- Can sort this in linear time.
- Get back 6.
- **6** is our new pivot!
Back to our original array! Use the pivot (=6) to break it up into two (well three).

We know the following:

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

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

\[
\begin{array}{cccccccccc}
4 & 3 & 7 & 1 & 9 & 10 & 8 & 11 & 2 & 12 & 6 & 5
\end{array}
\]

We know the following:

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

Call \text{Median-of-Medians}(A_{\text{Upper}}, 10 - 6 = 4)
Review linear time selection

Then we do this again:

| 7 | 9 | 10 | 8 | 11 | 12 |
Review linear time selection

Then we do this again:

\[
\begin{array}{cccccc}
7 & 9 & 10 & 8 & 11 & 12 \\
\end{array}
\]

First we reorganize:
Review linear time selection

Then we do this again:

| 7 | 9 | 10 | 8 | 11 | 12 |

First we reorganize:

<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Then we sort each column:

<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>
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

- Call Median-of-Medians([9, 12], floor(len/2) = 1)
- Can sort this in linear time.
- Get back 12.
- **12** is our new pivot!
Back to our original array! Use the pivot (=6) to break it up into two (well three).

```
7 9 10 8 11 12
```

We know the following:

- $\text{len}(A_{\text{Lower}}) = 5$
- $\text{len}(A_{\text{Upper}}) = 0$
- 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).

\[
\begin{array}{cccccc}
7 & 9 & 10 & 8 & 11 & 12 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 9 & 10 & 8 & 11 \\
\end{array} \\
\begin{array}{c}
12 \\
\end{array}
\]

We know the following:

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

Call \textbf{Median-of-Medians}(A_{\text{Lower}}, 4)
Review linear time selection

Final Step!

Can sort in linear time!

Return $Sorted(A[4]) = 11$
Median of medians time analysis

**Median-of-medians** \((A, i)\):

- sublists = \([A[j:j+5] \text{ for } j \in \text{range}(0, \text{len}(A), 5)]\)
- medians = \([\text{sorted} (\text{sublist})[\text{len} (\text{sublist})/2] \text{ for } \text{sublist} \in \text{sublists}]\)

// Base Case
if \(\text{len} (A) \leq 5\) return \(\text{sorted} (a)[i]\)

// Find median of medians
if \(\text{len} (\text{medians}) \leq 5\)
    pivot = \(\text{sorted} (\text{medians})[\text{len} (\text{medians})/2]\)
else
    pivot = Median-of-medians (medians, \(\text{len}/2\))

// Partitioning Step
low = \([j \text{ for } j \in A \text{ if } j < \text{pivot}]\)
high = \([j \text{ for } j \in A \text{ if } j > \text{pivot}]\)

\(k = \text{len} (\text{low})\)
if \(i = k\)
    return Median-of-medians (low, \(i\))
elseif \(i > k\)
    return Median-of-medians (low, \(i-k-1\))
else
    return pivot

\(T(n) = T(\frac{n}{10}) + T(\frac{n}{5}) + O(n)\)
Median of medians time analysis

Median-of-medians\((A, i)\):
```
sublists = [A[j:j+5] for j in range(0, len(A), 5)]
medians = [sorted(sublist)[len(sublist)/2] for sublist in 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 in A if j < pivot]
high = [j for j in 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
```

\[
T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + cn \equiv O(n)
\]
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?

\[ T(n) = T\left(\frac{5}{6}n\right) + T\left(\frac{2}{3}n\right) + cn \]

for (subarray \( l = 5 \))

Find median of median

Sort subarrays

\[ T(n) = T\left(\frac{4}{6}n\right) + T\left(\frac{1}{3}n\right) + cn \]

for (subarray \( l = 3 \))

Recursion on the main function

NOM > 1/6 elements

1/2 columns
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?

\[ T(n) = T\left(\frac{1}{3}n\right) + T\left(\frac{4}{6}n\right) + cn \]
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?

\[ T(n) = T\left(\frac{1}{3}n\right) + T\left(\frac{4}{6}n\right) + cn \]

What about \( k = 7 \)?

What about \( k = 7 \)?

\[ T(n) = T\left(\frac{1}{7}n\right) + T\left(\frac{10}{14}n\right) + cn \]

\[ \alpha(n) \]

\[ \alpha(n) \]
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?

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

What about $k = 7$?

$$T(n) = T\left(\frac{1}{7}n\right) + T\left(\frac{10}{14}n\right) + cn$$
On different techniques for recursive algorithms
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
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?

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

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

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

<table>
<thead>
<tr>
<th>N</th>
<th>Number of Solutions</th>
<th>Number of Unique Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>352</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>724</td>
<td>92</td>
</tr>
<tr>
<td>11</td>
<td>2,680</td>
<td>341</td>
</tr>
<tr>
<td>12</td>
<td>14,200</td>
<td>1,787</td>
</tr>
<tr>
<td>13</td>
<td>73,712</td>
<td>9,233</td>
</tr>
<tr>
<td>14</td>
<td>365,596</td>
<td>45,752</td>
</tr>
<tr>
<td>15</td>
<td>2,279,184</td>
<td>285,053</td>
</tr>
</tbody>
</table>
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**.
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 \(A\)

```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: \(O(n^2)\).

2. \(n\) subsequences of a sequence of length \(n\) and \(O(n)\) time to check if a given sequence is increasing.
Naive Enumeration

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

\[
\text{algLISNaive}(A[1..n]):
max = 0
\]
\[
\text{for each subsequence } B \text{ of } A \text{ do }
\]
\[
\quad \text{if } B \text{ is increasing and } |B| > max \text{ then }
\]
\[
\quad \quad max = |B|
\]

Output \( max \)

Running time: \( O(n \cdot 2^n) \)
Naive Enumeration

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

\[
\text{algLISNaive}(A[1..n]):
\]
\[
\begin{align*}
&\quad \text{max} = 0 \\
&\quad \text{for each subsequence } B \text{ of } A \text{ do} \\
&\quad \quad \text{if } B \text{ is increasing and } |B| > \text{max} \text{ then} \\
&\quad \quad \quad \text{max} = |B| \\
&\quad \text{Output } \text{max}
\end{align*}
\]

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?

\( \text{LIS}(A[1..n]):\)

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

\[ \{6, 3, \ldots\} \]
Recursive Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

\[
\text{LIS}(A[1..n]):
\]

- **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
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.
Can we find a recursive algorithm for LIS?

\[ \text{LIS}(A[1..n]) : \]

- **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.

**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 \(\text{LIS\_smaller}(A[1..n], x)\) which gives the longest increasing subsequence in \(A\) where each number in the sequence is less than \(x\).
Example

Sequence: \( A[1..5] = 5, 9, 7, 8, 1 \)

\[ ss = \square \]
\[ x = \text{inf} \]

\[ ss = \square \]
\[ x = \text{inf} \]

\[ ss = \square \]
\[ x = \text{inf} \]

\[ ss = [1] \]
\[ x = 1 \]

\[ ss = [7] \]
\[ x = 7 \]

\[ ss = [8] \]
\[ x = 8 \]

\[ ss = [78] \]
\[ x = 7 \]
Recursive Approach

\textbf{LIS\textunderscore smaller}(A[1..n], x) : length of longest increasing subsequence in } A[1..n] \text{ with all numbers in subsequence less than } x

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
LIS(A[1..n]):
    return LIS\_smaller(A[1..n], \infty)
\end{verbatim}
Running time analysis
Running time of \(\text{LIS}([1..n])\)

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

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

\[
A = \{1, 2, 3, 4, 5, 6\}
\]
Lemma\n\textbf{LIS\_smaller} runs in $O(2^n)$ time.
Lemma

\textbf{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!