Backtracking and Memoization

Lecture 12

Tuesday, October 4, 2022

LaTeXed: October 13, 2022 14:17

12.1
On different techniques for recursive algorithms

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 $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n-1}$ or less.
2. For termination, problem instances of small size are solved by some other method as **base cases**.

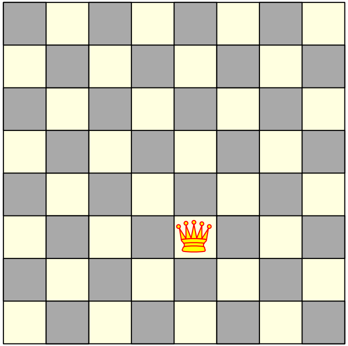
Recursion in Algorithm Design

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.

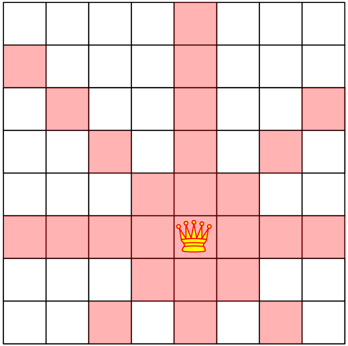
   Examples: Closest pair, deterministic median selection, quick sort.
3. **Backtracking**: Refinement of brute force search. Build solution incrementally by invoking recursion to try all possibilities for the decision in each step.
4. **Dynamic Programming**: problem reduced to multiple (typically) dependent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

12.2
Search trees and backtracking

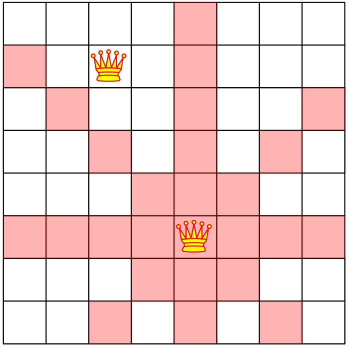
The queens problem



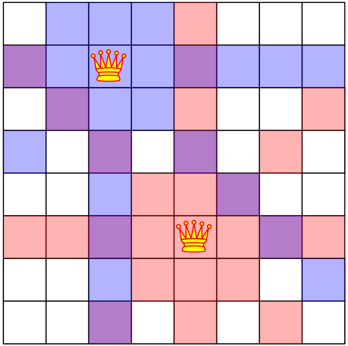
The queens problem



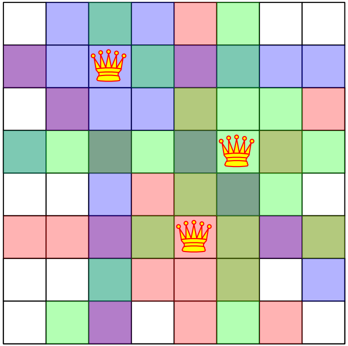
The queens problem



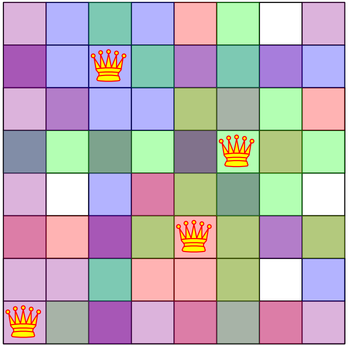
The queens problem



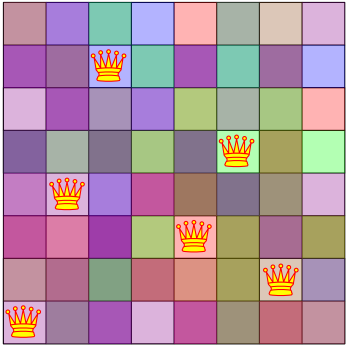
The queens problem



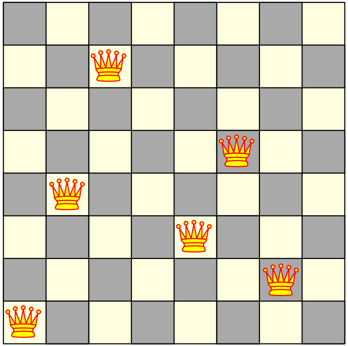
The queens problem



The queens problem



The queens problem

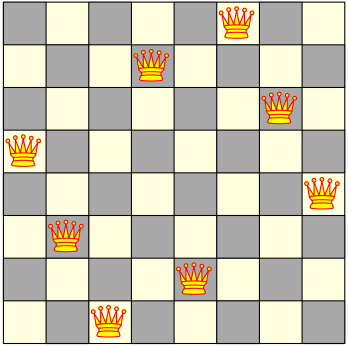


Q: How many queens can one place on the board?

Q: Can one place 8 queens on the board?

The eight queens puzzle

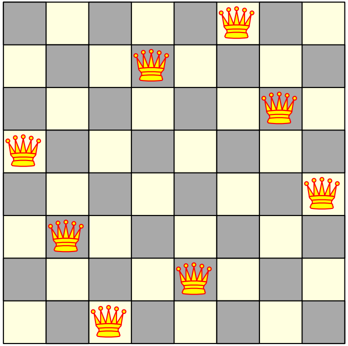
Problem published in 1848, solved in 1850.



Q: How to solve problem for general **n**?

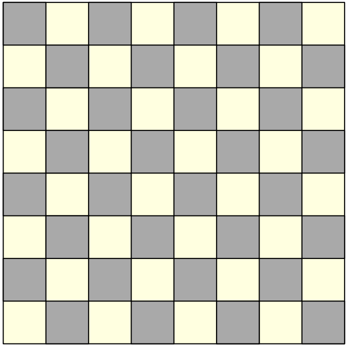
The eight queens puzzle

Problem published in 1848, solved in 1850.

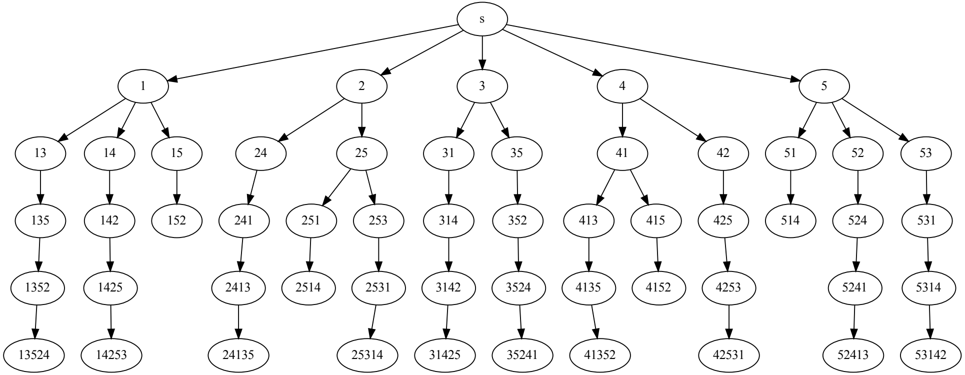


Q: How to solve problem for general **n**?

Strategy: Search tree



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 );
```

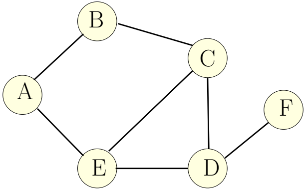
12.3 Brute Force Search, Recursion and Backtracking

12.3.1
Naive algorithm for Max Independent Set in a Graph

Maximum Independent Set in a Graph

Definition 12.1.

Given undirected graph $\mathbf{G = (V, E)}$ a subset of nodes $\mathbf{S} \subseteq \mathbf{V}$ is an independent set (also called a stable set) if for there are no edges between nodes in $\mathbf{S}$. That is, if $\mathbf{u, v} \in \mathbf{S}$ then $\mathbf{(u, v)} \notin \mathbf{E}$.

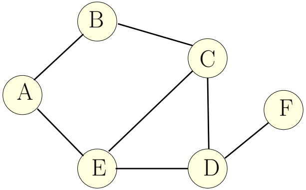


Some independent sets in graph above: $\mathbf{\{D\}, \{A, C\}, \{B, E, F\}}$

Maximum Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$

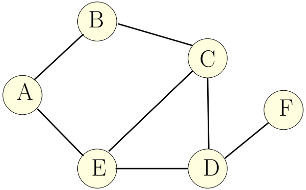
Goal Find maximum sized independent set in $\mathbf{G}$



Maximum Weight Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$, weights $\mathbf{w(v) \geq 0}$ for $\mathbf{v \in V}$

Goal Find maximum weight independent set in $\mathbf{G}$



Maximum Weight Independent Set Problem

1. No one knows an efficient (polynomial time) algorithm for this problem
2. Problem is **NP-Complete** and it is believed that there is no polynomial time algorithm

Brute-force algorithm:

Try all subsets of vertices.

Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)

    Output max
```

Running time: suppose G has n vertices and m edges

1. 2^n subsets of V
2. checking each subset S takes $O(m)$ time
3. total time is $O(m2^n)$

Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)

    Output max
```

Running time: suppose $\mathbf{G}$ has $\mathbf{n}$ vertices and $\mathbf{m}$ edges

1. $\mathbf{2^n}$ subsets of $\mathbf{V}$
2. checking each subset $\mathbf{S}$ takes $\mathbf{O(m)}$ time
3. total time is $\mathbf{O(m2^n)}$

12.3.2
A recursive algorithm for Max Independent Set in a Graph

A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex u let $N(u)$ be its neighbors.

Observation 12.2.

v_1: *vertex in the graph.*
One of the following two cases is true

Case 1 v_1 *is in some maximum independent set.*

Case 2 v_1 *is in no maximum independent set.*

We can try both cases to "reduce" the size of the problem

A Recursive Algorithm

Let $\mathbf{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$.
For a vertex $\mathbf{u}$ let $\mathbf{N(u)}$ be its neighbors.

Observation 12.2.

$\mathbf{v_1}$: *vertex in the graph.*
One of the following two cases is true

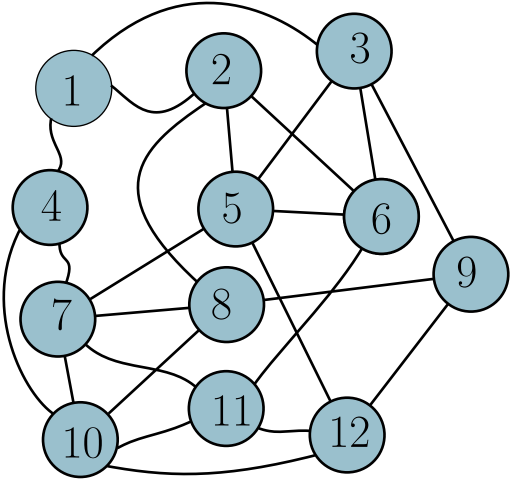
Case 1 $\mathbf{v_1}$ *is in <u>some</u> maximum independent set.*

Case 2 $\mathbf{v_1}$ *is in <u>no</u> maximum independent set.*

We can try <u>both cases</u> to "reduce" the size of the problem

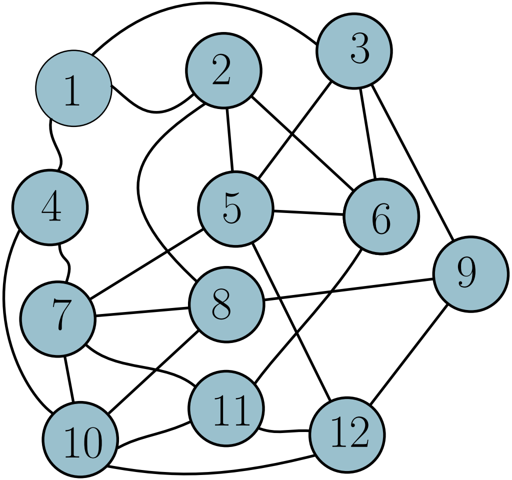
Removing a vertex (say 5)

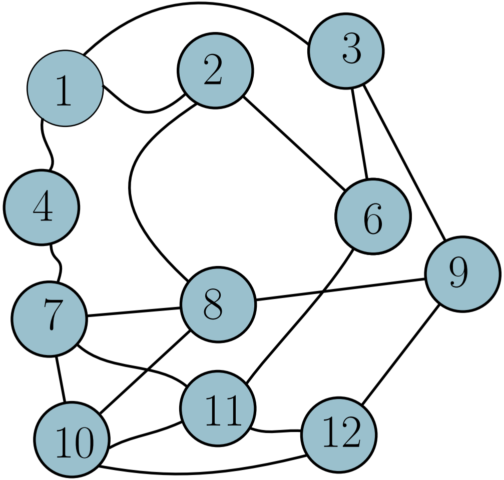
Because it is NOT in the independent set



Removing a vertex (say 5)

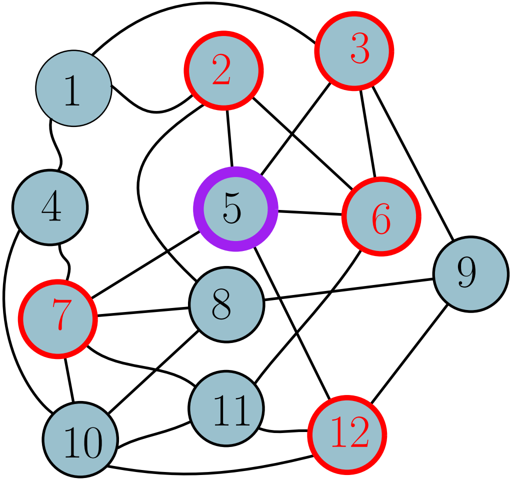
Because it is NOT in the independent set





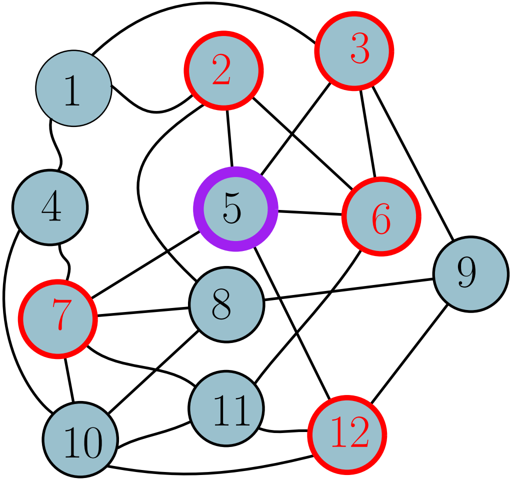
Removing a vertex (say 5) and its neighbors

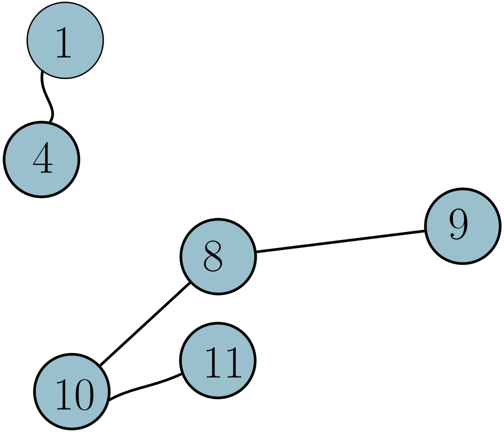
Because it **is** in the independent set



Removing a vertex (say 5) and its neighbors

Because it **is** in the independent set





A Recursive Algorithm: The two possibilities

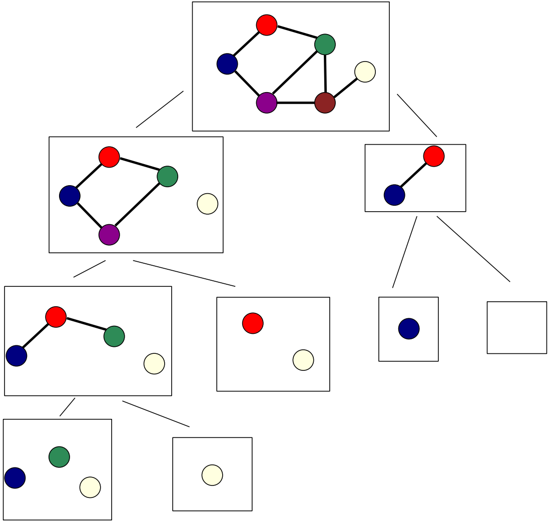
$\mathbf{G_1 = G - v_1}$ obtained by removing $\mathbf{v_1}$ and incident edges from $\mathbf{G}$
$\mathbf{G_2 = G - v_1 - N(v_1)}$ obtained by removing $\mathbf{N(v_1) \cup v_1}$ from $\mathbf{G}$

$$\mathbf{MIS(G) = \max\{MIS(G_1), MIS(G_2) + w(v_1)\}}$$

A Recursive Algorithm

```
RecursiveMIS(G):
    if G is empty then Output 0
    a = RecursiveMIS(G − v1)
    b = w(v1) + RecursiveMIS(G − v1 − N(vn))
    Output max(a, b)
```

Example



Recursive Algorithms

..for Maximum Independent Set

Running time:

$$\mathbf{T(n) = T(n-1) + T\Big(n - 1 - deg(v_1)\Big) + O(1 + deg(v_1))}$$

where $\mathbf{deg(v_1)}$ is the degree of $\mathbf{v_1}$. $\mathbf{T(0) = T(1) = 1}$ is base case.

Worst case is when $\mathbf{deg(v_1) = 0}$ when the recurrence becomes

$$\mathbf{T(n) = 2T(n-1) + O(1)}$$

Solution to this is $\mathbf{T(n) = O(2^n)}$.

Backtrack Search via Recursion

1. Recursive algorithm generates a tree of computation where each node is a smaller problem (subproblem)
2. Simple recursive algorithm computes/explores the whole tree blindly in some order.
3. Backtrack search is a way to explore the tree intelligently to prune the search space
    3.1 Some subproblems may be so simple that we can stop the recursive algorithm and solve it directly by some other method
    3.2 Memoization to avoid recomputing same problem
    3.3 Stop the recursion at a subproblem if it is clear that there is no need to explore further.
    3.4 Leads to a number of heuristics that are widely used in practice although the worst case running time may still be exponential.

12.4
Longest Increasing Subsequence

Sequences

Definition 12.1.

Sequence: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

Definition 12.2.

$a_{i_1}, \ldots, a_{i_k}$ is a **subsequence** of $a_1, \ldots, a_n$ if $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

Definition 12.3.

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 12.4.

1. Sequence: $\mathbf{6, 3, 5, 2, 7, 8, 1, 9}$
2. Subsequence of above sequence: $\mathbf{5, 2, 1}$
3. Increasing sequence: $\mathbf{3, 5, 9, 17, 54}$
4. Decreasing sequence: $\mathbf{34, 21, 7, 5, 1}$
5. Increasing <u>subsequence</u> of the first sequence: $\mathbf{2, 7, 9}$.

Longest Increasing Subsequence Problem

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

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

Example 12.5.

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

Longest Increasing Subsequence Problem

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

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

Example 12.5.

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

Naïve 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.

Naïve 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.

Naïve 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: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

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

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

Observation 12.6.

For second case we want to find a subsequence in $\mathbf{A[1..(n-1)]}$ *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**.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

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

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

Observation 12.6.

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

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

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

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

Observation 12.6.

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**.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

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

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

[redacted]

For second case we want to find a subsequence in $\mathbf{A[1..(n-1)]}$ *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**.

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], ∞)
```

Example

Sequence: $\mathbf{A[1..7] = 6, 3, 5, 2, 7, 8, 1}$

12.4.1
Running time analysis

Running time of LIS($\mathbf{[1..n]}$)

```
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($\mathbf{[1..n]}$)

Lemma 12.7.

LIS_smaller *runs in* $\mathbf{O(2^n)}$ *time.*

Improvement: From $\mathbf{O(n2^n)}$ to $\mathbf{O(2^n)}$.

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

Running time of LIS($\mathbf{[1..n]}$)

Lemma 12.7.

LIS_smaller *runs in* $\mathbf{O(2^n)}$ *time.*

Improvement: From $\mathbf{O(n2^n)}$ to $\mathbf{O(2^n)}$.

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

Running time of LIS($\mathbf{[1..n]}$)

Lemma 12.7.

LIS_smaller *runs in* $\mathbf{O(2^n)}$ *time.*

Improvement: From $\mathbf{O(n2^n)}$ to $\mathbf{O(2^n)}$.

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