COMP1110/1140/6710
Structured programming

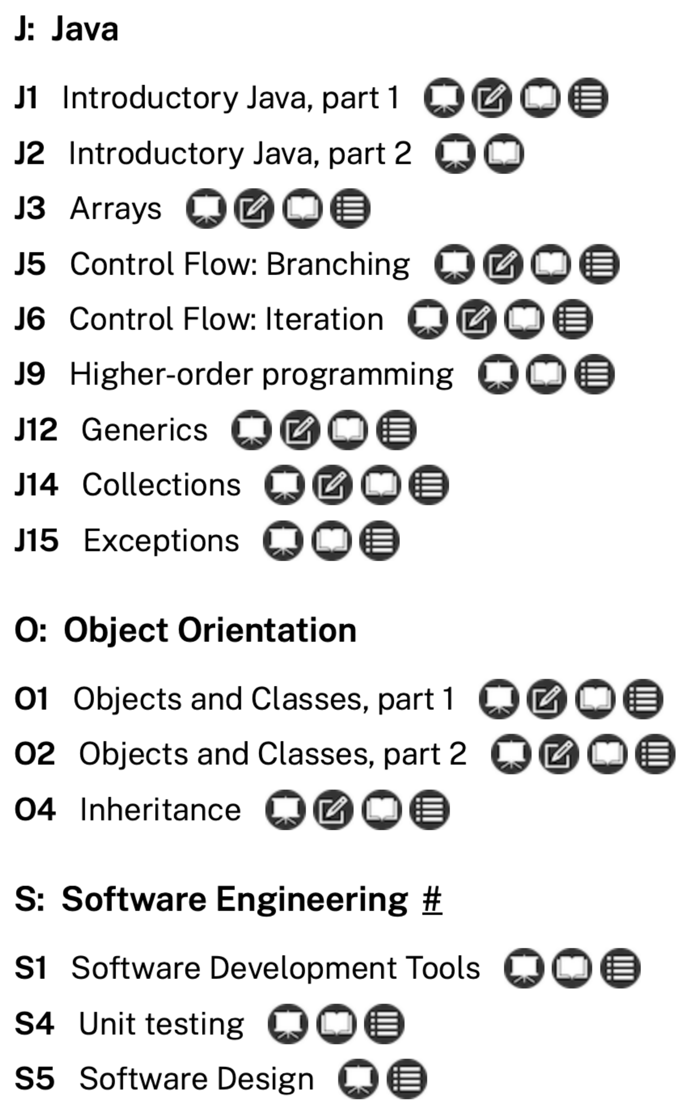
Week-7 (A)

Recursion Revisited

16-Sept-2024

Topics for the **first half** of this semester (H1):

- Java language features
- Object-Oriented programming
- Code Development & Software Engineering



J: Java

J1 Introductory Java, part 1
J2 Introductory Java, part 2
J3 Arrays
J5 Control Flow: Branching
J6 Control Flow: Iteration
J9 Higher-order programming
J12 Generics
J14 Collections
J15 Exceptions

O: Object Orientation

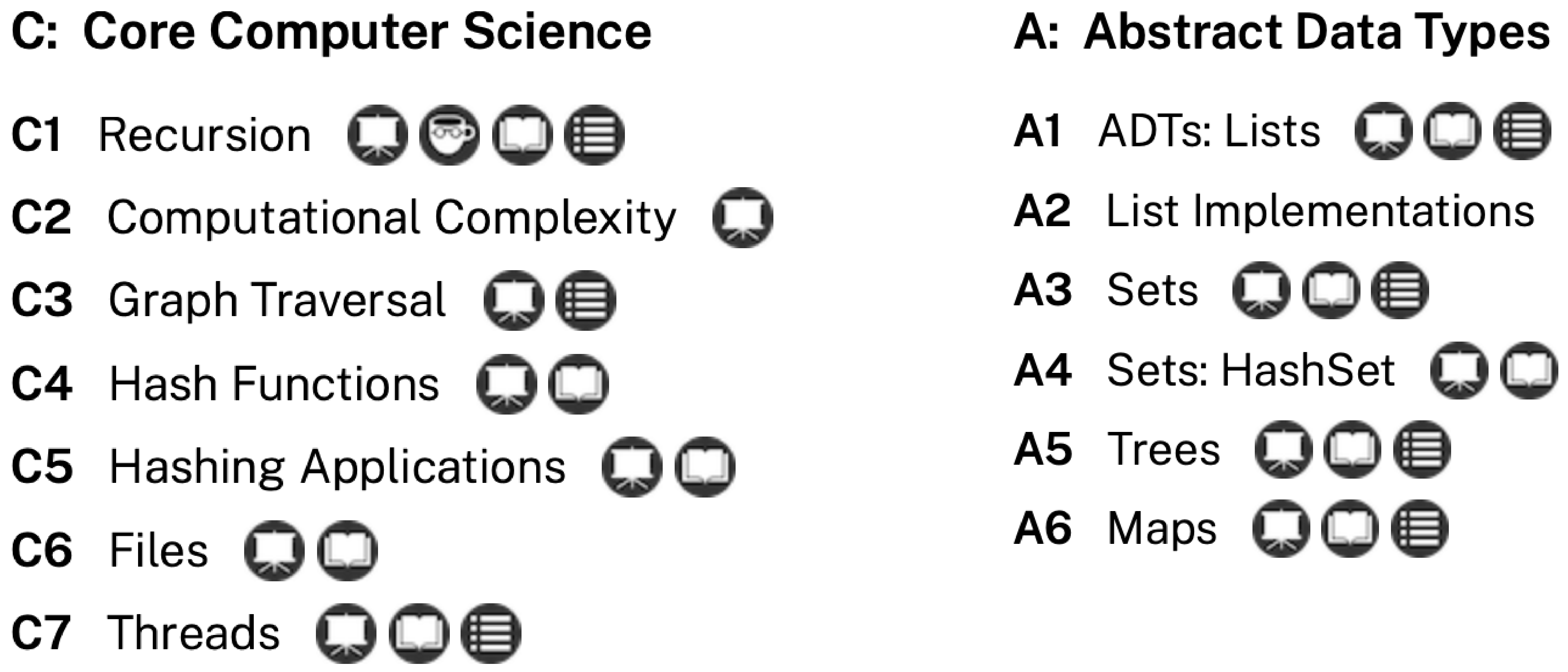
O1 Objects and Classes, part 1
O2 Objects and Classes, part 2
O4 Inheritance

S: Software Engineering #

S1 Software Development Tools
S4 Unit testing
S5 Software Design

Topics for the **second half** of this semester (H2):

Core Computer Science & ADT:



C: Core Computer Science

C1 Recursion

C2 Computational Complexity

C3 Graph Traversal

C4 Hash Functions

C5 Hashing Applications

C6 Files

C7 Threads

A: Abstract Data Types

A1 ADTs: Lists

A2 List Implementations

A3 Sets

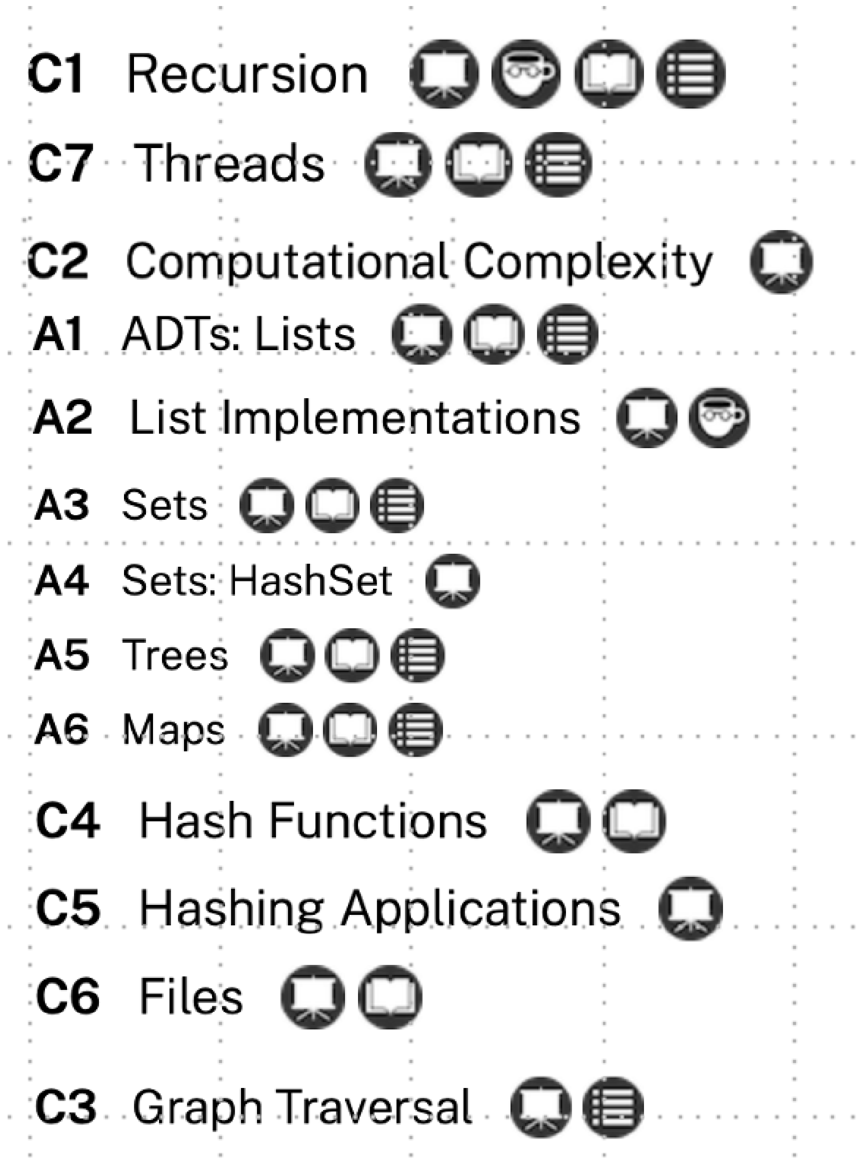
A4 Sets: HashSet

A5 Trees

A6 Maps

Outline (tentative plan) for 2H of 2024 S2

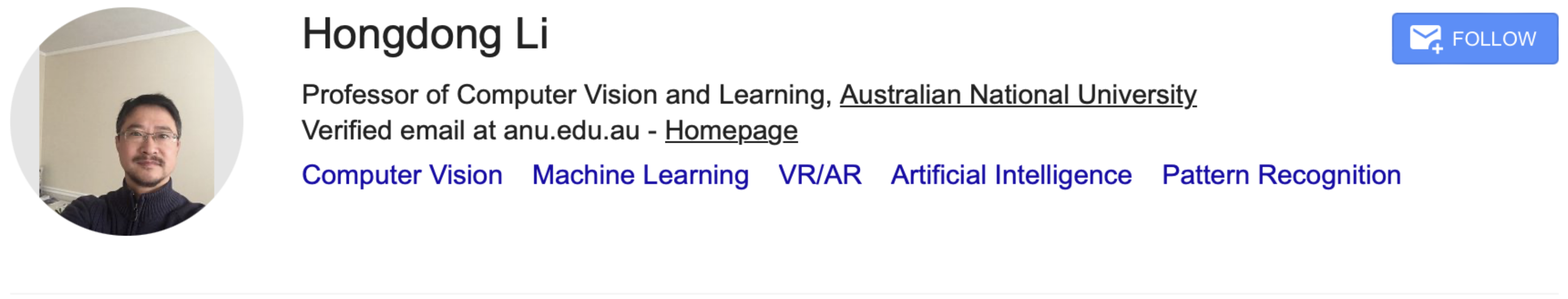
- Week-7: (HL)
  - Recursion revisited (deep dive)
  - Backtracking technique
  - (optional) Java: multi-thread (TBC)
  - Complexity Analysis and Big-O notation
- Week-8: (HL)
  - ADT : Basic Concepts
  - ADT List, and list Implementations
  - Array and Linked-list
- Week-9: (PH)
  - ADT: trees
  - Binary search trees
  - ADT: set and map
  - Java: files
- Week-10: (PH)
  - ADT: Hash table, hash set
  - Hash code and hash function
  - Hash applications
  - Software system development.
- Week-11: (HL)
  - ADT: graph data structure
  - Graph traversal (DPS, BFS, A*)
  - Graph algorithms (MST, minimal path, TSP, max-flow min-cut).
- Week-12: Class Review (PH)



C1 Recursion
C7 Threads
C2 Computational Complexity
A1 ADTs: Lists
A2 List Implementations
A3 Sets
A4 Sets: HashSet
A5 Trees
A6 Maps
C4 Hash Functions
C5 Hashing Applications
C6 Files
C3 Graph Traversal

A few words about your 2nd convenor/lecturer: Hongdong Li

- https://scholar.google.com/citations?user=Mq89JAcAAAAJ&hl=en



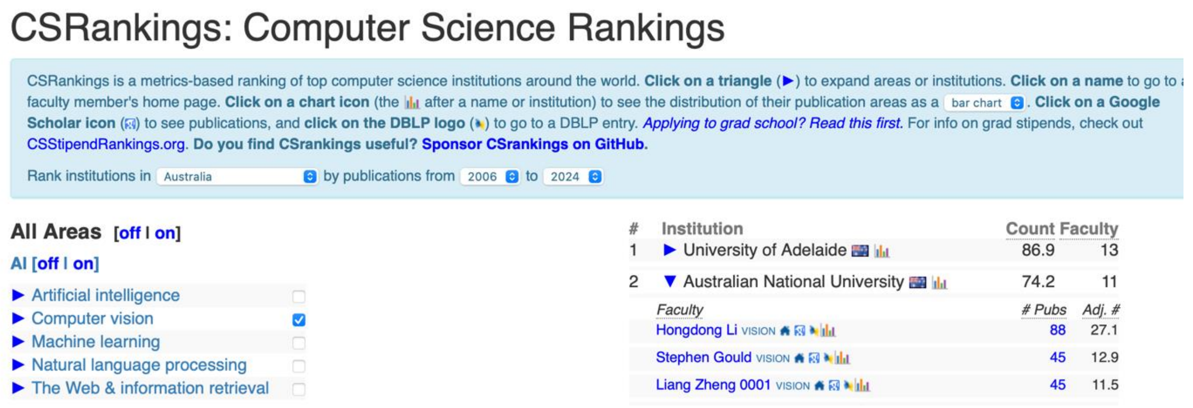
Hongdong Li

FOLLOW

Professor of Computer Vision and Learning, Australian National University
Verified email at anu.edu.au - Homepage

Computer Vision   Machine Learning   VR/AR   Artificial Intelligence   Pattern Recognition

- CSRanking:
- https://csrankings.org/#/fromyear/2006/toyear/2024/index?vision&au



CSRankings: Computer Science Rankings

CSRankings is a metrics-based ranking of top computer science institutions around the world. **Click on a triangle** (▶) to expand areas or institutions. **Click on a name** to go to
faculty member's home page. **Click on a chart icon** (the 📊 after a name or institution) to see the distribution of their publication areas as a bar chart. **Click on a Google Scholar icon** to see publications, and **click on the DBLP logo** to go to a DBLP entry. *Applying to grad school? Read this first.* For info on grad stipends, check out CSStipendRankings.org. **Do you find CSrankings useful? Sponsor CSrankings on GitHub.**

Rank institutions in Australia by publications from 2006 to 2024

All Areas [off | on]

AI [off | on]

- ▶ Artificial intelligence ☐
- ▶ Computer vision ☑
- ▶ Machine learning ☐
- ▶ Natural language processing ☐
- ▶ The Web & information retrieval ☐

#	Institution	Count	Faculty
1	▶ University of Adelaide	86.9	13
2	▼ Australian National University	74.2	11
	Faculty	*# Pubs*	*Adj. #*
	Hongdong Li VISION	88	27.1
	Stephen Gould VISION	45	12.9
	Liang Zheng 0001 VISION	45	11.5

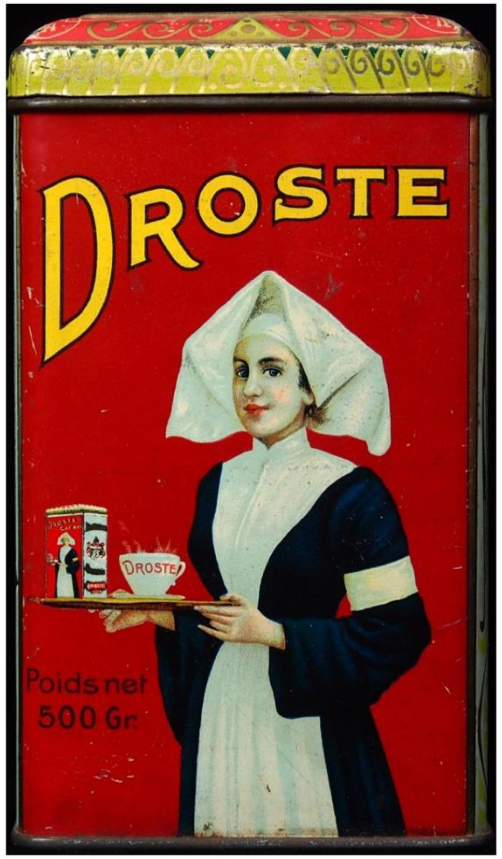
This week’s topics

- (A): Recursion revisited (deep dive)
- (B) : [Optional] Java: multi-threads and parallel programming
- (C) : Computational Complexity

Recursion Revisited

Recall:
Recursion (the basics)

- "...embedding an expression of some type within an expression of the same type" (*linguistics*)



(Alf van Beem, CC0, via Wikimedia Commons)

Recursion (the basics)

- The body of a function can contain function calls, including *calls to the same function*.
  - This is known as recursion.
- The function must have a branching statement, such that a recursive call does not always take place (“base case”); otherwise, recursion never ends.
- Recursion is a way to think about solving a problem: how to reduce it to a simpler instance of itself?

Learning Objectives

- Thinking recursively
- Tracing execution of a recursive method
- Writing recursive algorithms
- Recursive data structures
    - E.g. LinkedList, tree, etc.

Quotes about “recursion”

- *“To iterate is human, to recurse, divine.”*

  L. Peter Deutsch, computer scientist, or

  Robert Heller, computer scientist, or …

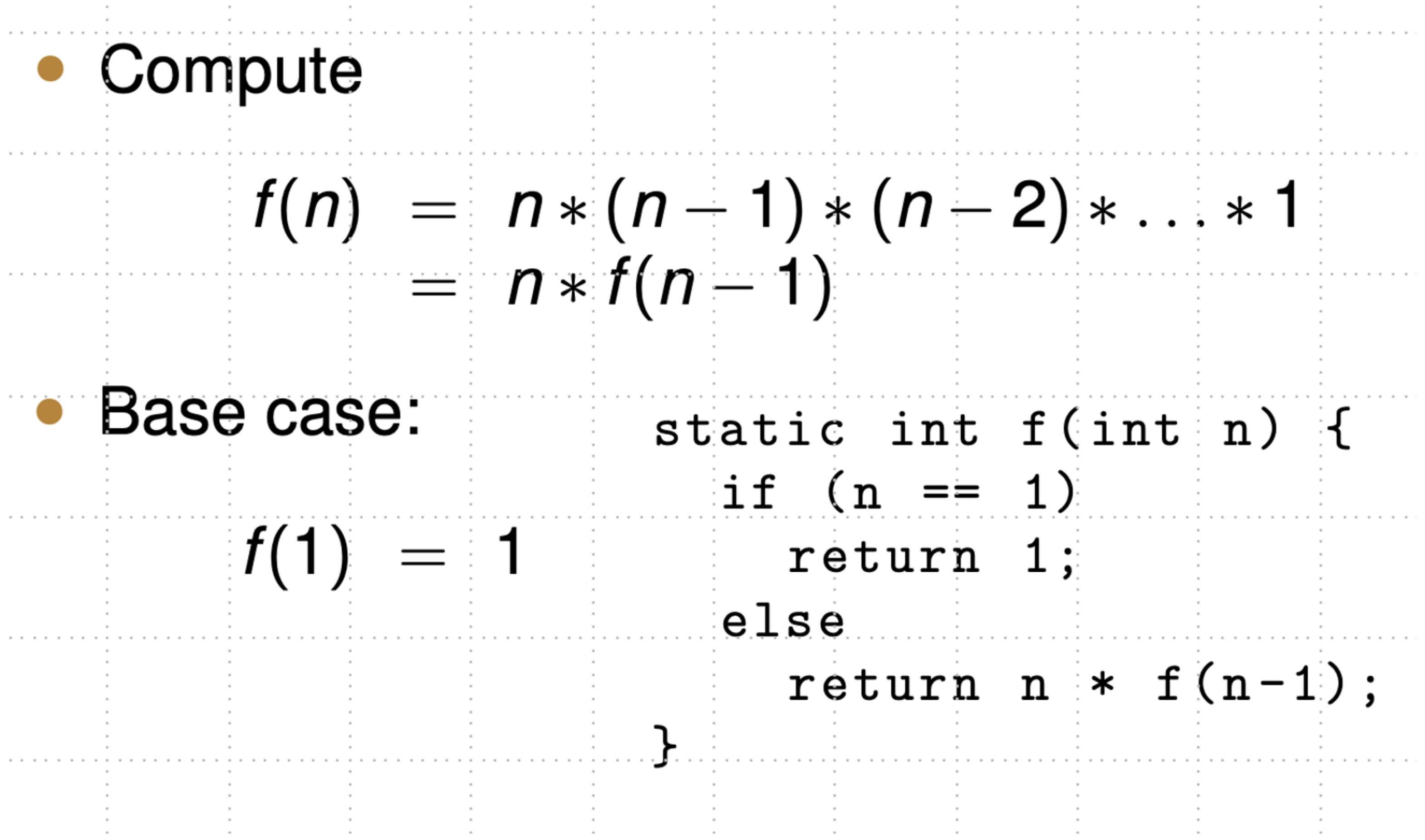
- *“Mastering the recursive way of thinking is what separates a Computer Scientist from all other people who also know computer programming. ”*

Recursive Thinking

- **Recursion** is:
  - A problem-solving **approach**, that can …
  - Generate simple solutions to …
  - Certain kinds of problems that …
  - Would be difficult to solve in other ways
- **Recursion** splits a problem:
  - Into one or more simpler versions of **itself**

Recursive Thinking: Example(1):

Factorial function



- Compute

$$\begin{aligned} f(n) &= n * (n-1) * (n-2) * \ldots * 1 \\ &= n * f(n-1) \end{aligned}$$

- Base case:

$$f(1) = 1$$

```
static int f(int n) {
  if (n == 1)
    return 1;
  else
    return n * f(n-1);
}
```

Recursive Thinking: Example(2)

Strategy for processing nested dolls: (Matryoshka dolls)

```
1.  if there is only one doll
2.          do what it needed for it
    else
3.          do what is needed for the outer doll
4.          Process the inner nest in the same way
```

FIGURE 7.1
A Set of Nested Wooden Figures

Recursive Thinking: Example (3)

Strategy for binary searching a sorted array or list:

```
1. if the array is empty
2.       return -1 as the search result (not present)
3. else if the middle element == target
4.       return the index  of the middle element
5. else if target < middle element
6.       recursively search elements before middle point
7. else
8.       recursively search elements after the middle point
```

The General Approach

```
1. if problem is “small enough” and can be solved directly,
   i.e. the base case problem
2.     solve it directly
3. else
4.     break into one or more smaller subproblems
5.     solve each subproblem recursively
6.     combine results into solution to whole problem
```

Requirements for Recursion

- At least one "*small enough*" base case that you can solve directly
- A way of *breaking* a larger problem down into:
  - One or more *smaller* subproblems
  - Each of the *same kind* as the original
- A way of *combining* subproblem results into an overall solution to the larger problem

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

Recursion Design Strategy

- Identify the *base case(s)* (for direct solution)
- Devise a problem *splitting strategy*
  - Subproblems must be smaller
  - Subproblems must work towards a base case
- Devise a solution *combining strategy*

Recursion Design: Example

Recursive algorithm for finding the length of a string:

```
1.  if string is empty (no characters)
2.        return 0      ← base case
3.  else   ← recursive case
4.        compute length of string without the first character
5.        return 1 + that length
```

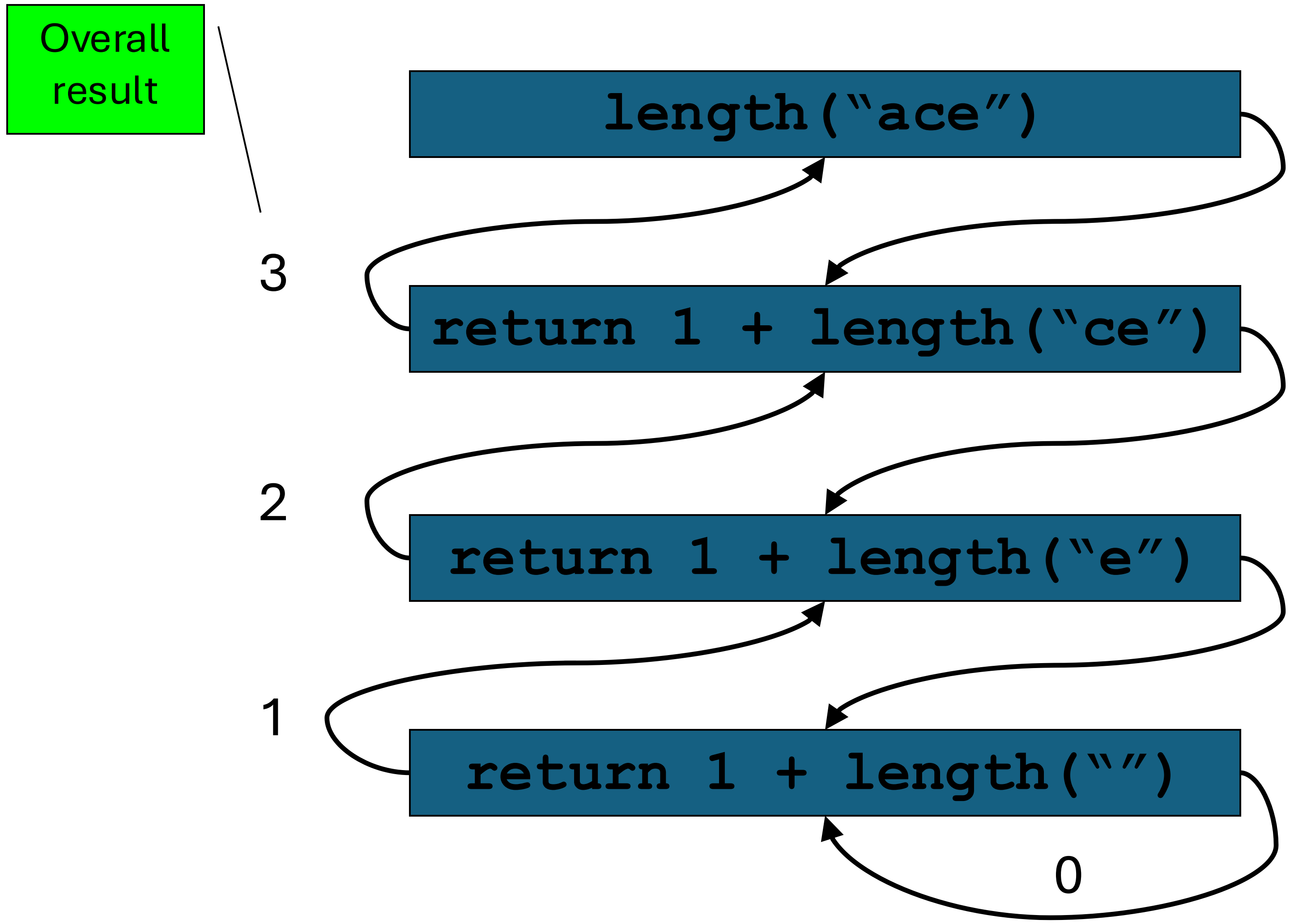
Note: Not the best technique for this problem.

Recursive Design Example: Java code

Recursive algorithm for finding the length of a string:

```
public static int length (String str) {
  if (str == null ||
       str.equals(“”))
     return 0;
  else
     return length(str.substring(1)) + 1;
}
```

Tracing a Recursive Method (1)



In class Exercise:

Write a recursive function isPalindrome accepts a string and returns true if it reads the same forwards as backwards.

```
isPalindrome("madam") → true
isPalindrome("racecar") → true
isPalindrome("step on no pets") → true
isPalindrome("Java") → false
isPalindrome("byebye") →false
```

Recall : three “musts” for recursion

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

The solution

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
bool isPalindrome(const string& s) {
    if (s.length() < 2) { // base case
        return true;
    } else { // recursive case
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
```

How to prove a Recursive Method is correct ?

Recursive proof is similar to math. induction:

1. Show *base case* recognized and *solved correctly*

2. Show that
   - ***If*** all *smaller problems* are *solved correctly*,
   - ***Then*** *original problem* is also *solved correctly*

3. Show that each recursive case makes progress towards the base case 🡐 *terminates properly*

Recursion and Mathematical Induction

- Recursion and mathematical induction
  - Both use a base case to solve a problem
  - Both solve smaller problems of the same type to derive a solution

- Induction can be used to
  - Prove properties about recursive algorithms
  - Prove that a recursive algorithm performs a certain amount of work

Proving the correctness by Mathematical Induction

1. Prove the theorem for the *base case(s):* ***n=0***

2. Show that:

   - ***If*** the theorem is *assumed true* for ***n***,
   - ***Then*** it *must be true* for ***n+1***

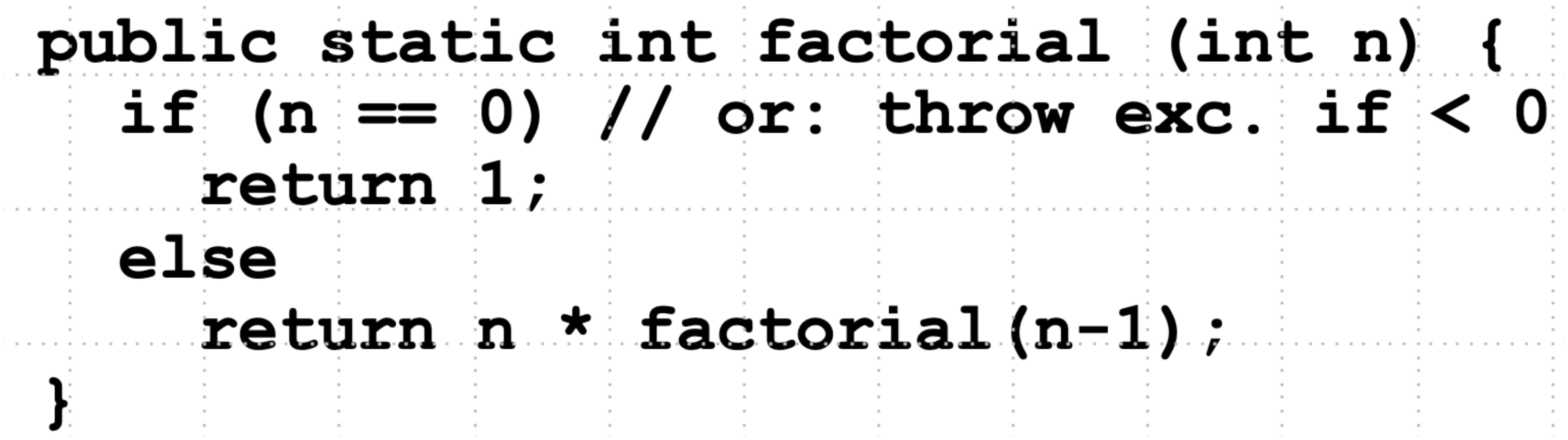
Result: Theorem true for all n ≥ 0.

Recursive Definition of some mathematical functions

- Mathematicians often use *recursive definitions*
- These lead very naturally to *recursive programs*
- Examples include:
  - Factorial
  - Power
  - GCD (Greatest common divisor)

Factorial (, once again)

- 0! = 1
- n! = n x (n-1)!



```
public static int factorial (int n) {
   if (n == 0) // or: throw exc. if < 0
      return 1;
   else
      return n * factorial(n-1);
}
```

- If a recursive function never reaches its base case, a stack overflow error occurs

Prove the Correctness of the Recursive Factorial

- Pseudocode for recursive factorial

```
if (n is 0)
    return 1
else
    return n * fact(n - 1)
```

- Induction on *n* proves the return values:
  - `fact(0) = 0! = 1`
  - `fact(n) = n!= n*(n - 1)*`
    `(n - 2)*…* 1 if n > 0`

Based on Mathematical Induction

Power function

- $x^0 = 1$
- $x^n = x \times x^{n-1}$

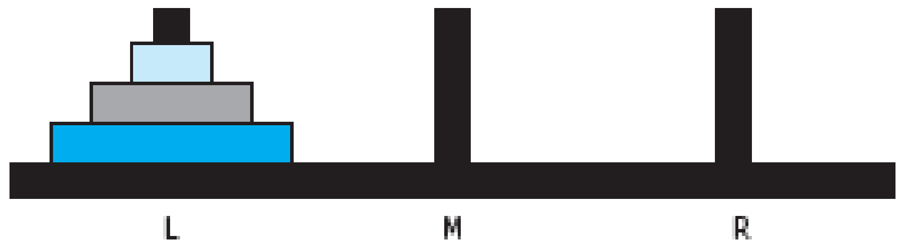
```
public static double power
    (double x, int n) {
  if (n <= 0)  // or: throw exc. if < 0
    return 1;
  else
    return x * power(x, n-1);
}
```

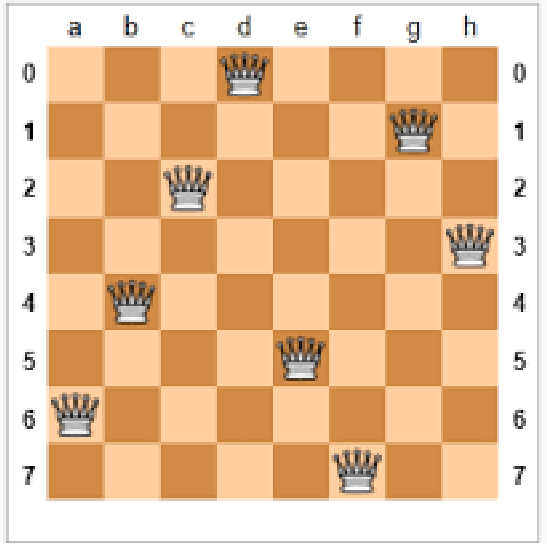
Problem Solving with Recursion

Problem Solving with Recursion

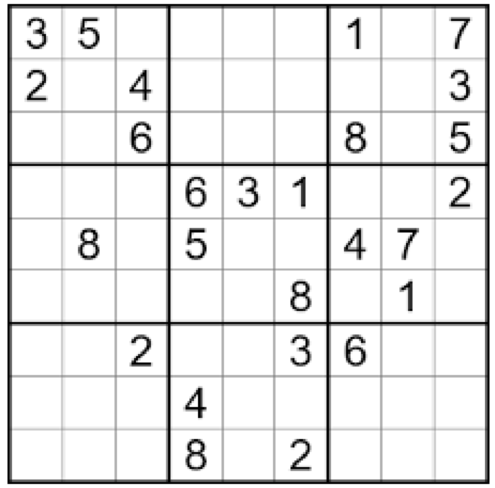
- Towers of Hanoi
- Backtracking
  - Maze puzzle
  - 8-Queens puzzle
  - Sudoku puzzle
  - …

FIGURE 7.11
Children's Version of Towers of Hanoi





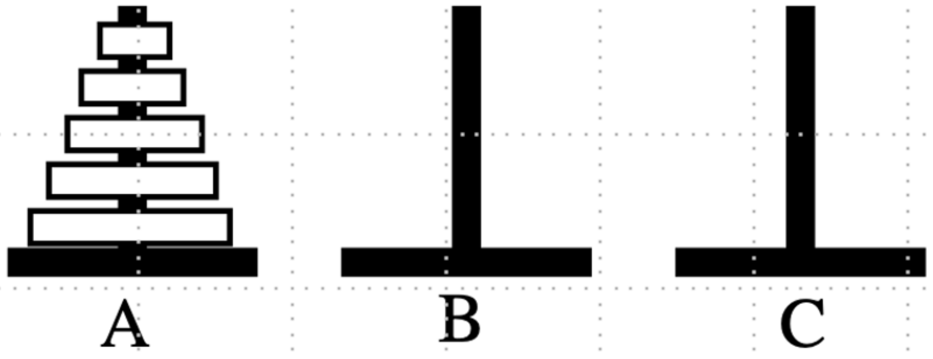
One solution to the eight queens puzzle



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

Towers of Hanoi: Description

(from *An introduction to Algorithms and Data Structures*, J. A. Storer, Springer, 2002)



Problem: You are given three posts labeled *A*, *B*, and *C*.

On Post *A* there are *n* rings of different sizes, in the order of the largest ring on the bottom to the smallest one on top.

Posts *B* and *C* are empty.

The object is to move the *n* rings from Post *A* to Post *B* by successively moving a ring from one post to another post that is empty or has a larger diameter ring on top.

Towers of Hanoi

- The ancient folklore :

“In the great temple at Benares, says he, beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed ***sixty-four discs*** *of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the* ***sixty-four discs*** *shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.''*

How to solve this puzzle ?

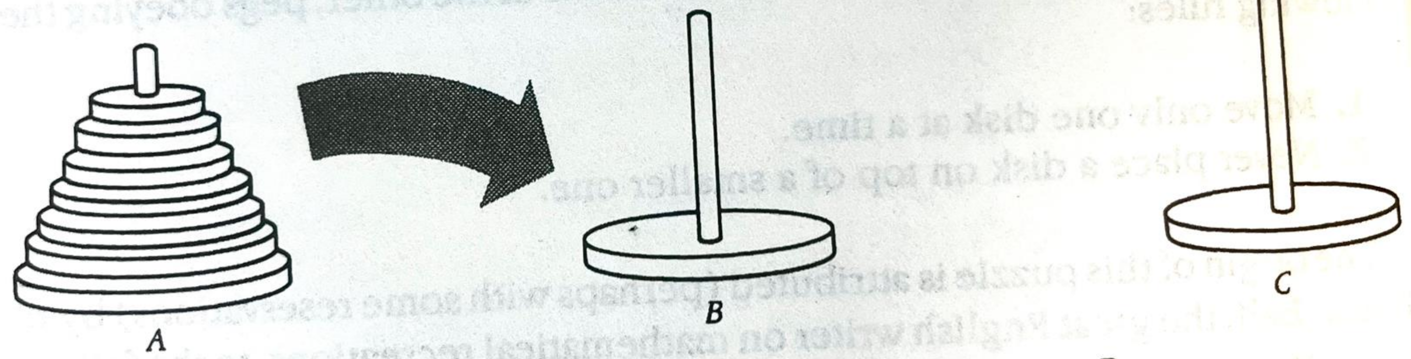
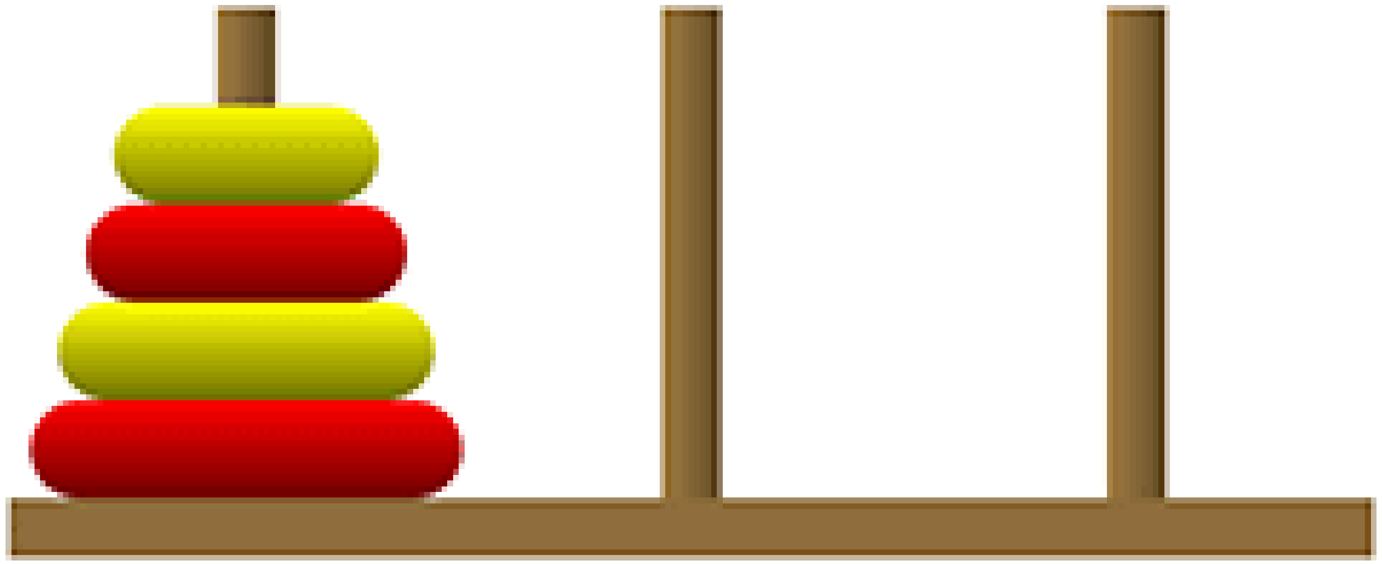


Figure 55.1 Move all the disks from peg A to peg B

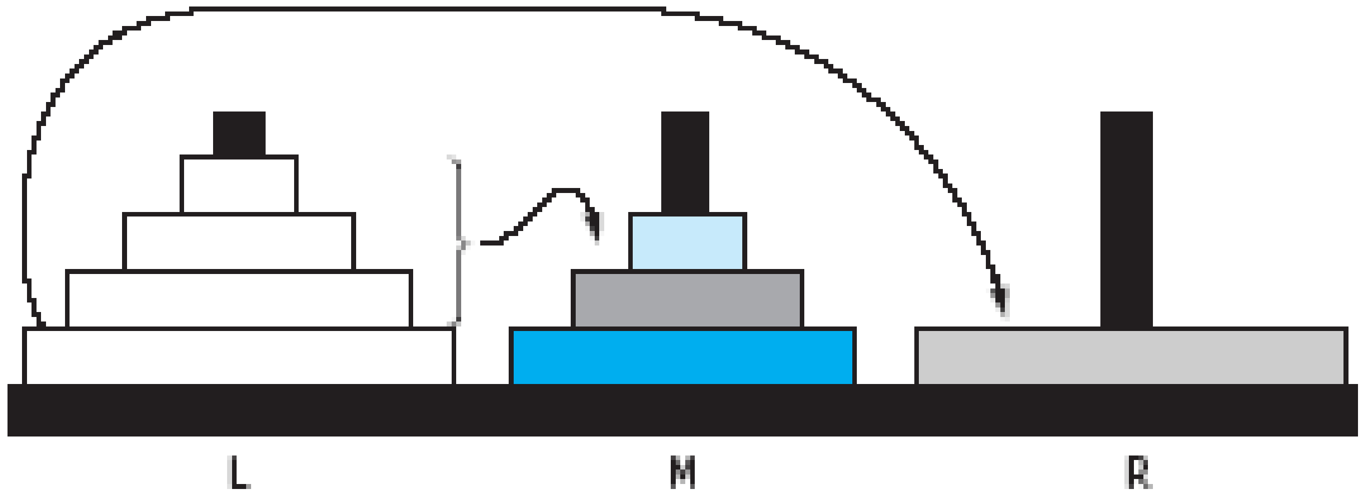
Example of 4 disks



A **Recursive** Solution Strategy

FIGURE 7.14

Towers of Hanoi After the First Two Steps in Solution of the Four-Disk Problem



Towers of Hanoi:

Recursion Structure

```
procedure: hanoi (n, A, B)
    if n = 1
    then move disk from A to B
    else hanoi (n − 1, A, C)
         move disk from A to B
         hanoi (n − 1, C, B)
```

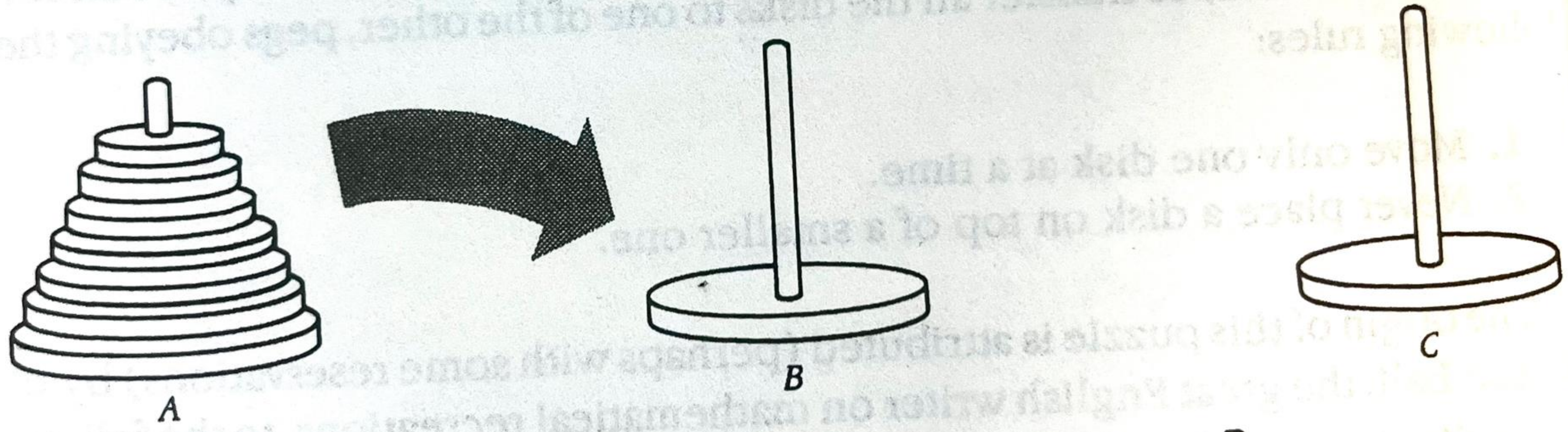


Figure 55.1 Move all the disks from peg A to peg B

Towers of Hanoi:

Java Code

```
public class TowersOfHanoi {

  public static String showMoves(int n,
        char src, char dst, char tmp) {

    if (n == 1)
      return “Move disk 1 from “ + src +
             “ to “ + dst + “\n”;

    else return

      showMoves(n-1, src, tmp, dst) +
      “Move disk “ + n + “ from “ + src +
        “ to “ + dst + “\n” +
      showMoves(n-1, tmp, dst, src);
  }
}
```

Performance Analysis:

a primitive example of “computational complexity analysis”

How many steps required to solve the Hanoi tower of size n?

We’ll just count lines; call this T(n).

- For n = 1, one line: $T(1) = 1$
- For n > 1, one line plus twice T(n) for next smaller size:

  $T(n+1) = 2 \times T(n) + 1$

Solving this gives $T(n) = 2^n - 1 = O(2^n)$

A formal proof of time complexity

Recurrence relation for the number of moves: For n=1, the two calls for n–1 do nothing and exactly one move is made. For n>1, twice whatever the number of moves required for n–1 are made plus the move made by the *write* statement. Hence, the number of moves made by TOWER on input n is:

$$T(n) = \begin{cases} 1 \text{ if } n = 1 \\ 2T(n-1)+1 \text{ otherwise} \end{cases}$$

Theorem: For $n \geq 1$, $TOWER(n,x,y,z)$ makes 2^n–1 moves.

Proof:

For n=1:

$$T(1) = 1 = 2^1 - 1$$

Now assume that TOWER works correctly for all values in the range 0 to n–1. Then:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 1 \end{aligned}$$

Time-complexity when n= 64

- 2^64-1 = ?
- If one can precisely make one move per second by hands, it will take about **580 trillion years** to succeed.
- Age of our universe: the universe is estimated to be 13.7 billion years old.

Warning: Don’t even try this for very large n (say n= 64) on a computer ; you will do a lot of string concatenation and garbage collection, and then run out of computer memory and may crash your computer !

Recursion versus Iteration

- Recursion and iteration are *similar*
- **Iteration:**
  - Loop repetition test determines whether to exit
- **Recursion:**
  - Condition tests for a base case
- Can always write iterative solution to a problem solved recursively, *but:*
- Recursive code often simpler than iterative
  - Thus, is easier to write, read, and understand.
  - However, the memory complexity can be huge (may use up func call stack very quickly.)

Recursive versus Iterative Methods

All recursive algorithms/methods
can be rewritten without recursion.

- Iterative methods use loops instead of recursion
- Iterative methods generally run faster and use less memory--less overhead in keeping track of method calls

Tail Recursion → Iteration

When recursion involves *single call* that is *at the end* …

It is called ***tail recursion*** and it easy to make iterative:

```
public static int iterFact (int n) {
  int result = 1;
  for (int k = 1; k <= n; k++) {
    result = result * k;
  }
  return result;
}
```

Efficiency of Recursion

- Recursive method often *slower* than iterative; ***why?***
    - Overhead for loop repetition smaller than
    - Overhead for function call and return

- If easier to develop algorithm using recursion,
    - Then code it as a recursive method:
    - Software engineering benefit probably outweighs …
    - Reduction in efficiency

- Don't "optimize" prematurely!

So When Should You Use Recursion?

- Solutions/algorithms for some problems are inherently recursive
  - iterative implementation could be more complicated

- When efficiency is less important
  - it might make the code easier to understand

- Bottom line is about:
  - Algorithm design
  - Tradeoff between readability and efficiency

A preview:

Next week, we will try to solve the “Towers of Hanoi” puzzle with iteration, and analyze their computational complexity.

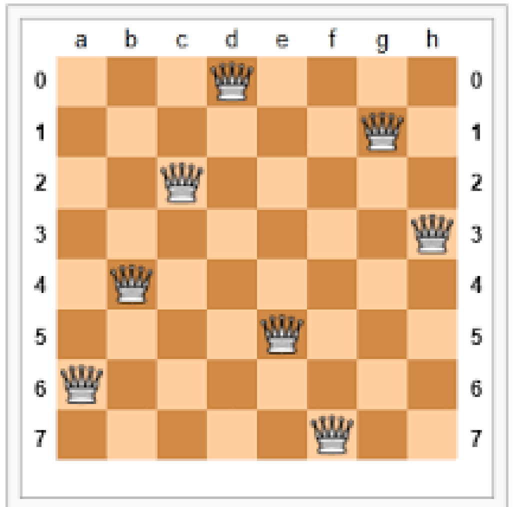
Recap: recursion

- **Recursion**
  - Break a problem into smaller subproblems of the same form, and call the same function again on that smaller form.
  - Super powerful programming tool
  - Not always the perfect choice, but often a good one
  - Some beautiful problems are solved recursively

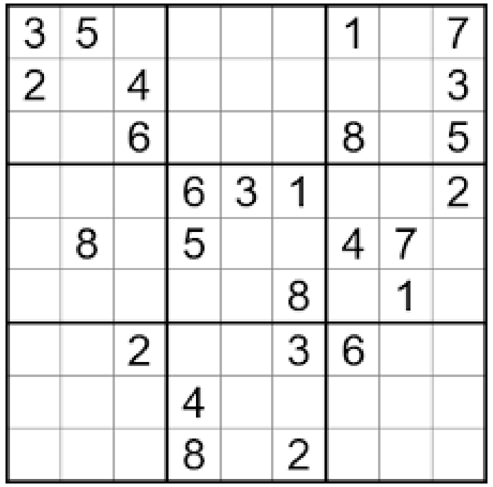
- **Three Musts for Recursion:**
  1. Your code must have a case for all valid inputs
  2. You must have a base case that makes no recursive calls
  3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

5-minute break

Backtracking for games



One solution to the eight queens puzzle



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

Backtracking technique

- Backtracking: A *systematic* trial-and-error search for solve a complex problem
  - *Examples:*
    - Finding a path through a maze
    - Board game (Chess, Go,..)
    - Sudoku puzzle
    - …
- For example, in walking through a maze, probably walk a path as far as you can go
    - Eventually, reach destination or a dead end
    - If dead end, must retrace your steps
    - Loops: stop when reach place you've been before
- Backtracking systematically tries alternative paths and eliminates them if they don't work

Backtracking technique (Cont.)

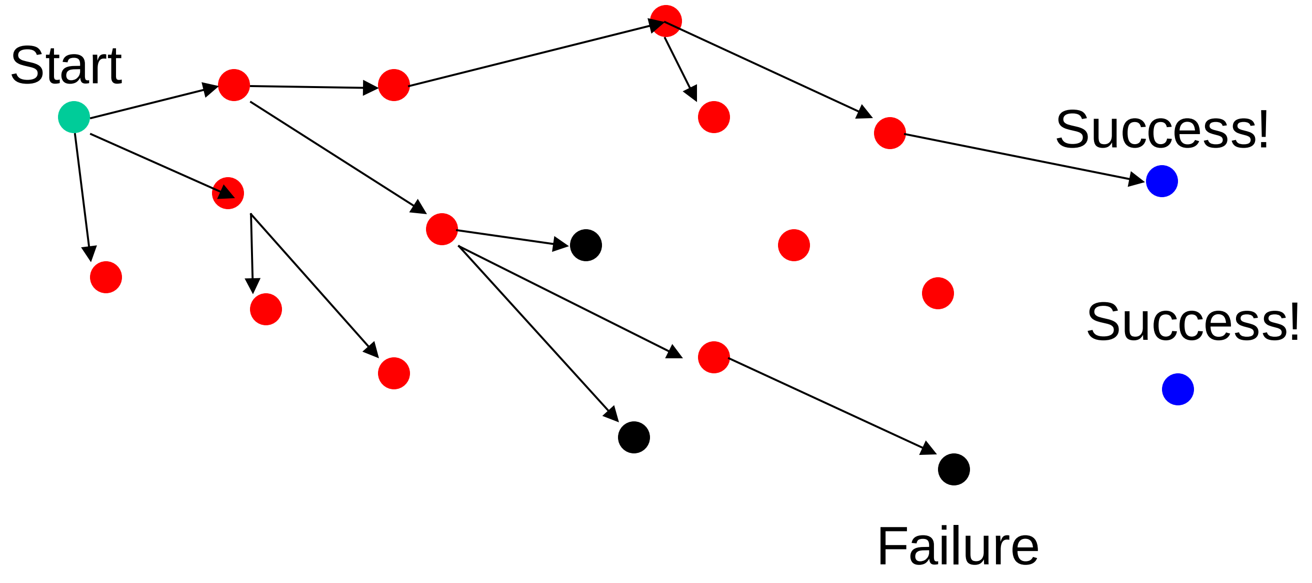
- If you never try exact same path more than once, and
- You try all possibilities,
- You will eventually find a solution if one exists

- Problems solved by backtracking: a set of *choices*

- Recursion implements backtracking straightforwardly
  - Activation frame remembers choice made at that decision point

- A chess playing program likely involves backtracking

Backtracking

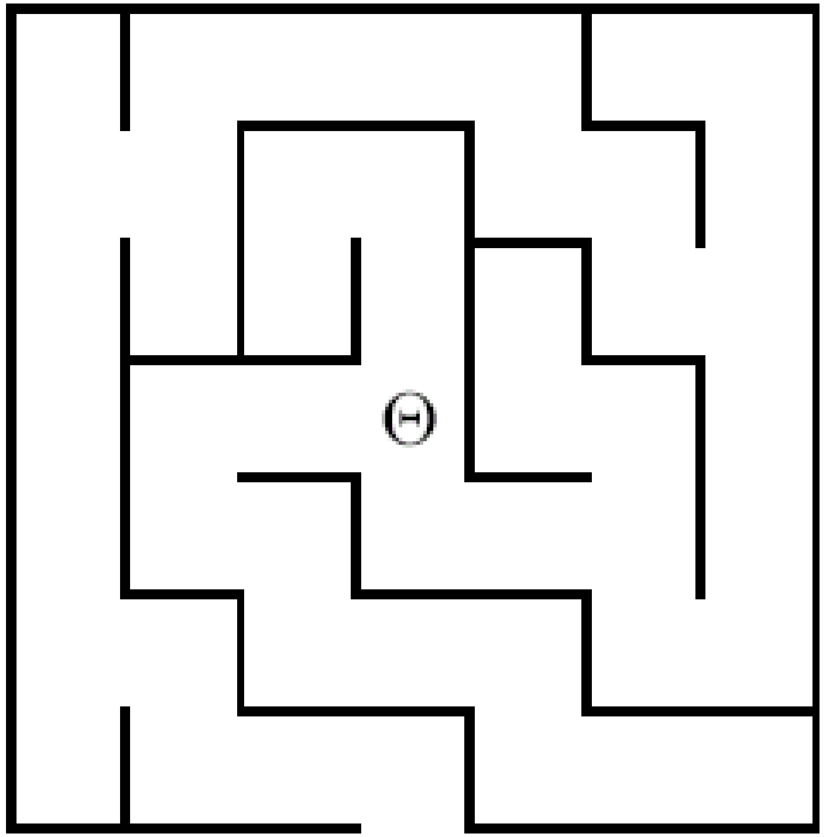


- Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can only see paths to connected nodes

- If a node only leads to failure go back to its "parent” node. Try other alternatives. If these all lead to failure then more backtracking may be necessary.

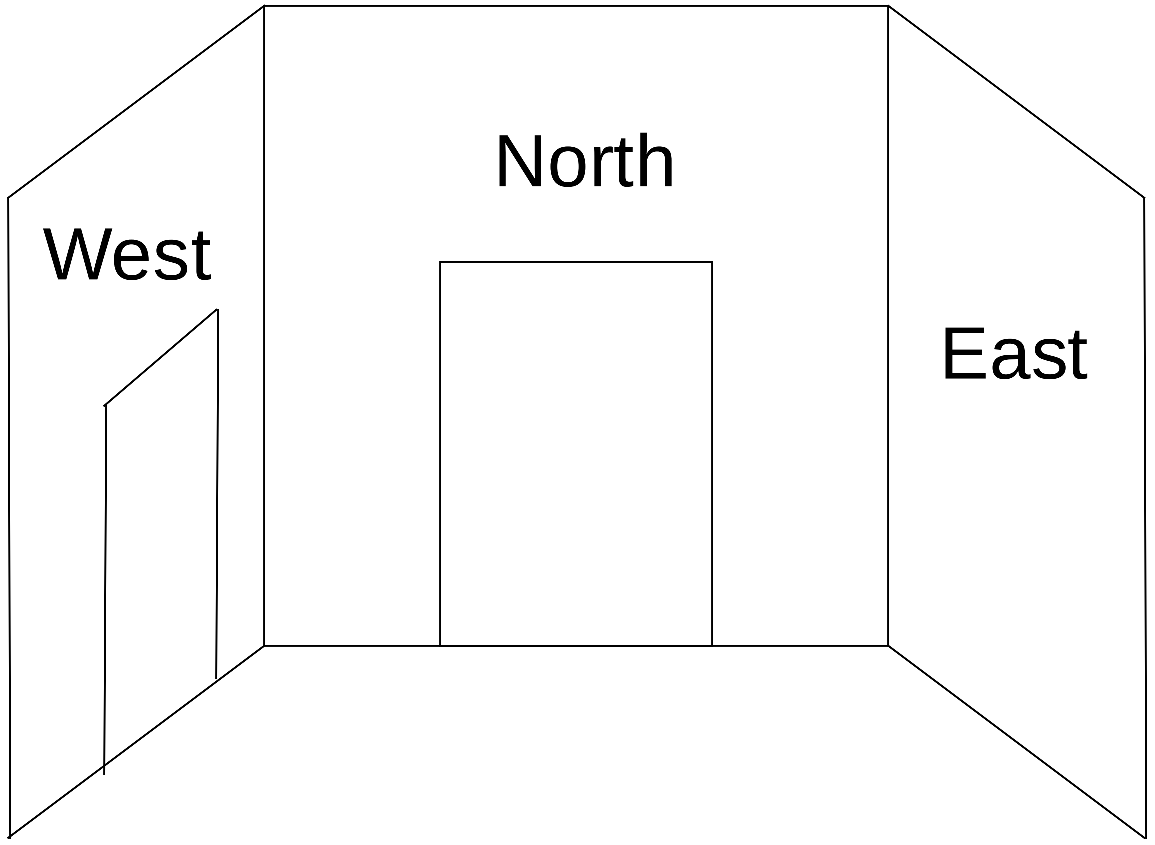
slide adapted from Recursive Backtracking by Mike Scott, UT Austin

Example:
A Simple Maze



Search maze until way out is found. If no way out possible, report that.

The Local View



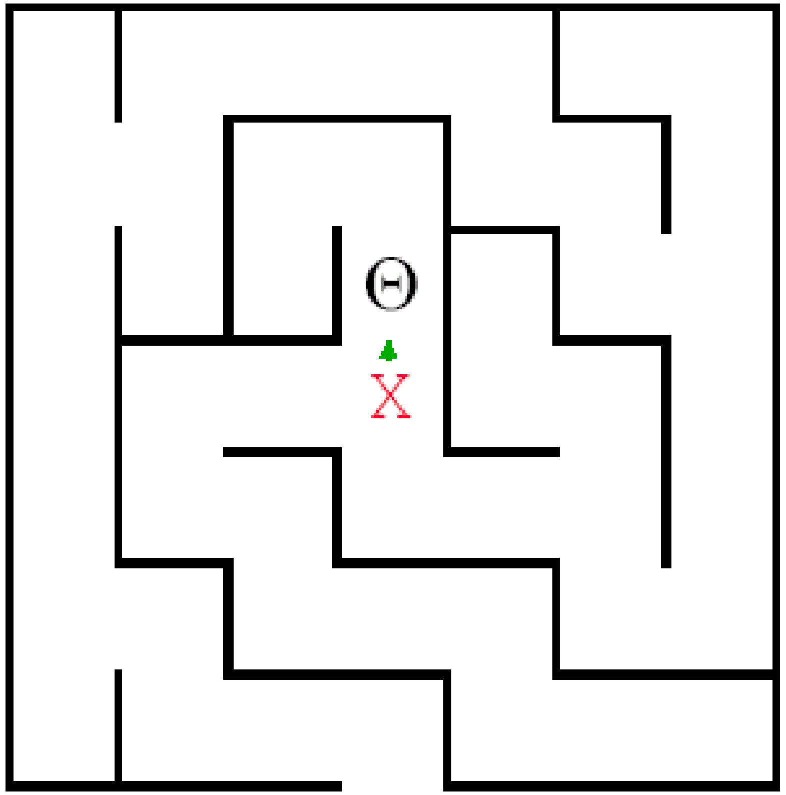
Which way do I go to get out?

Behind me, to the South is a door leading South

Backtracking Algorithm for Maze

```
If the current square is outside, return TRUE to indicate that
a solution has been found.
If the current square is marked, return FALSE to indicate that
this path has been tried.
Mark the current square.
for (each of the four compass directions)
{    if ( this direction is not blocked by a wall )
     {       Move one step in the indicated direction from the
current square.
             Try to solve the maze from there by making a
recursive call.
             If this call shows the maze to be solvable, return
TRUE to indicate that fact.
     }
}
Unmark the current square.
Return FALSE to indicate that none of the four directions led
to a solution.
```

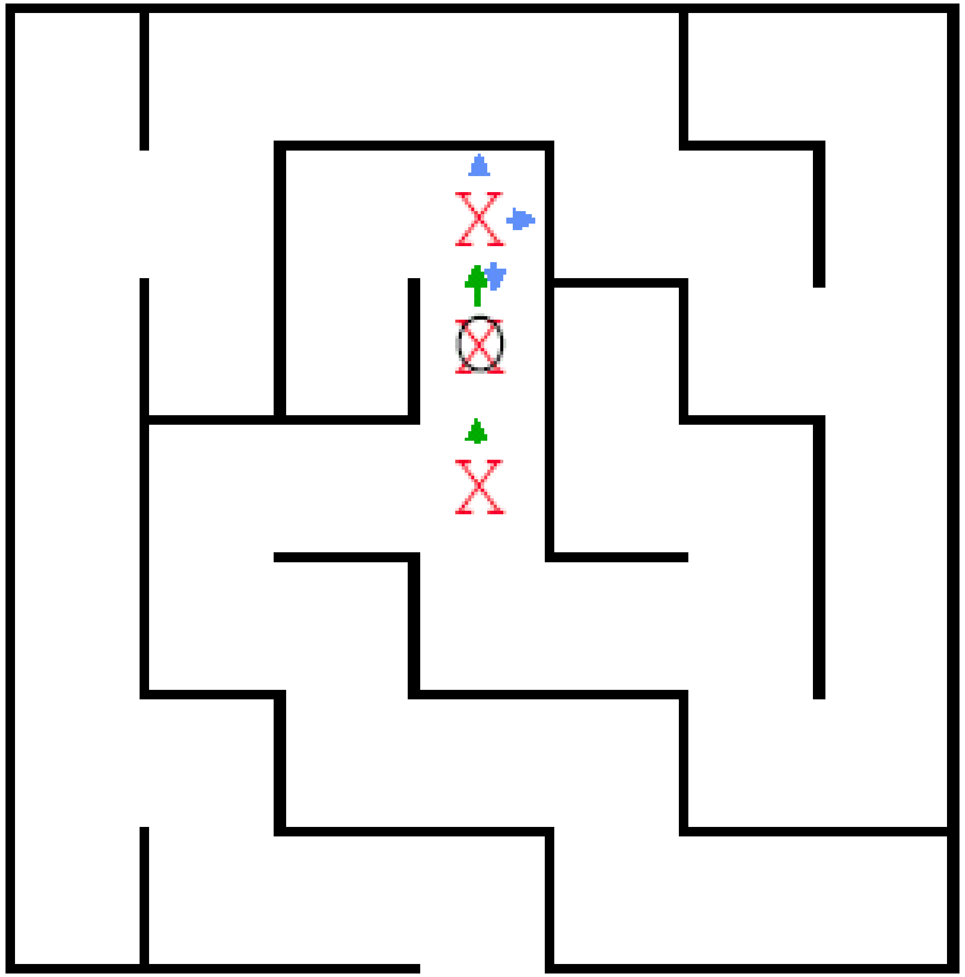
Backtracking in Action



The crucial part of the algorithm is the for loop that takes us through the alternatives from the current square. Here we have moved to the North.

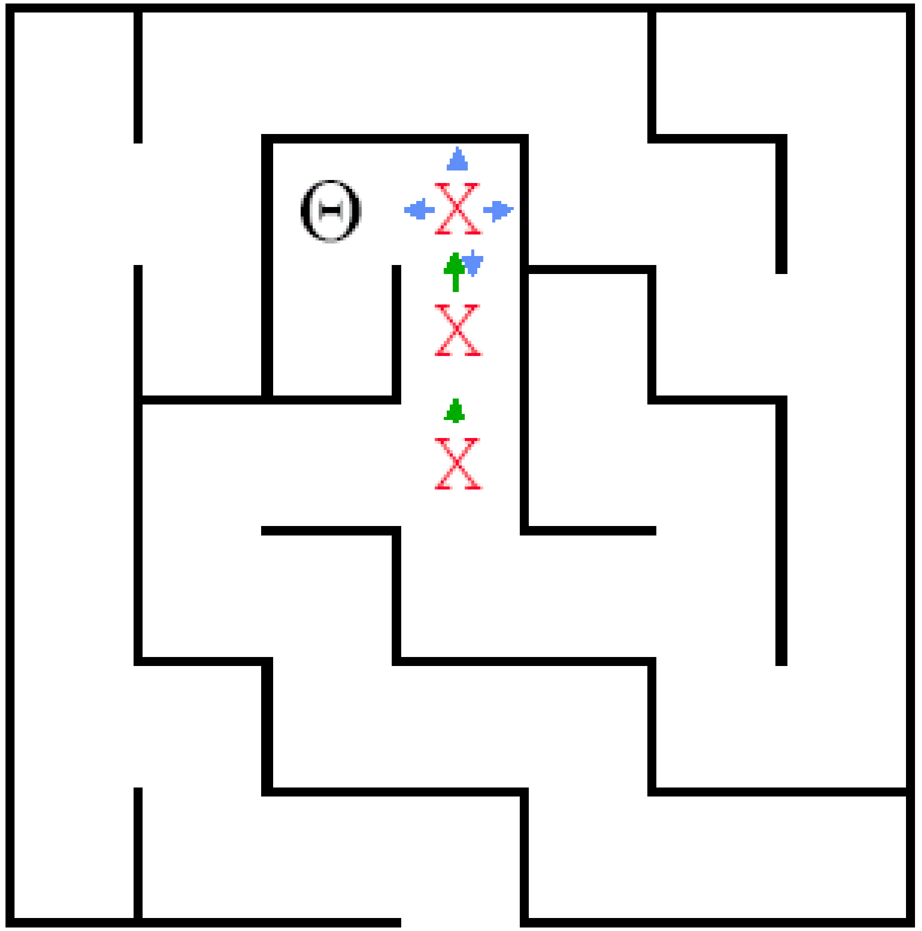
```
for (dir = North; dir <= West; dir++)
{      if (!WallExists(pt, dir))
{if (SolveMaze(AdjacentPoint(pt, dir)))
       return(TRUE);
}
```

Backtracking in Action



Here we have moved North again, but there is a wall to the North . East is also blocked, so we try South. That call discovers that the square is marked, so it just returns.

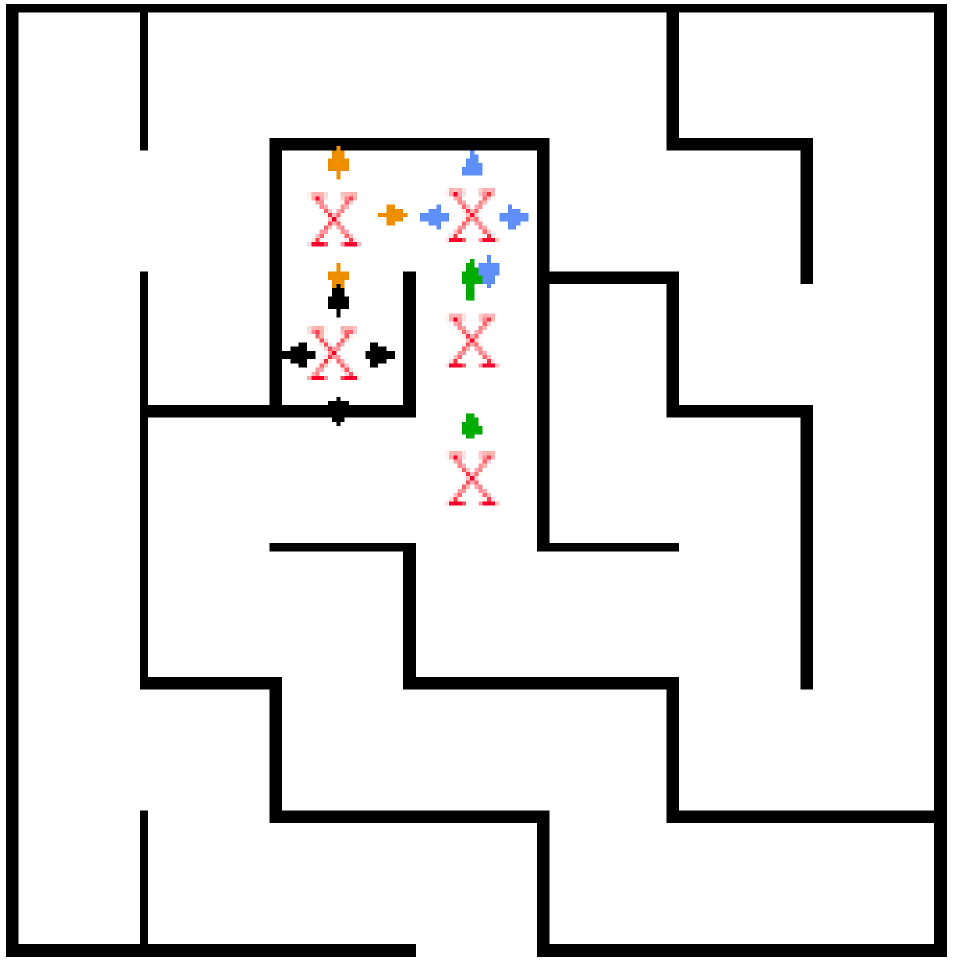
Backtracking in Action



So the next move we can make is West.

Where is this leading?

Backtracking in Action

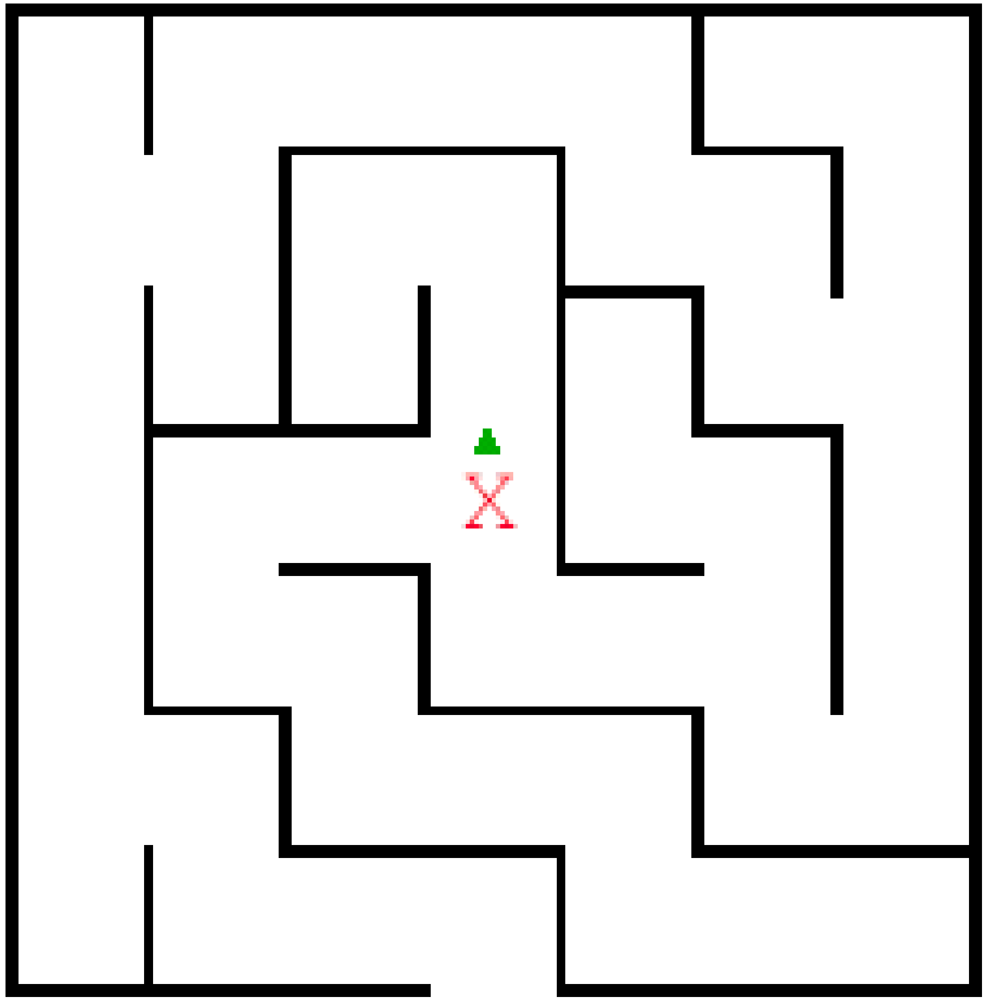


This path reaches a dead end.

Time to backtrack!

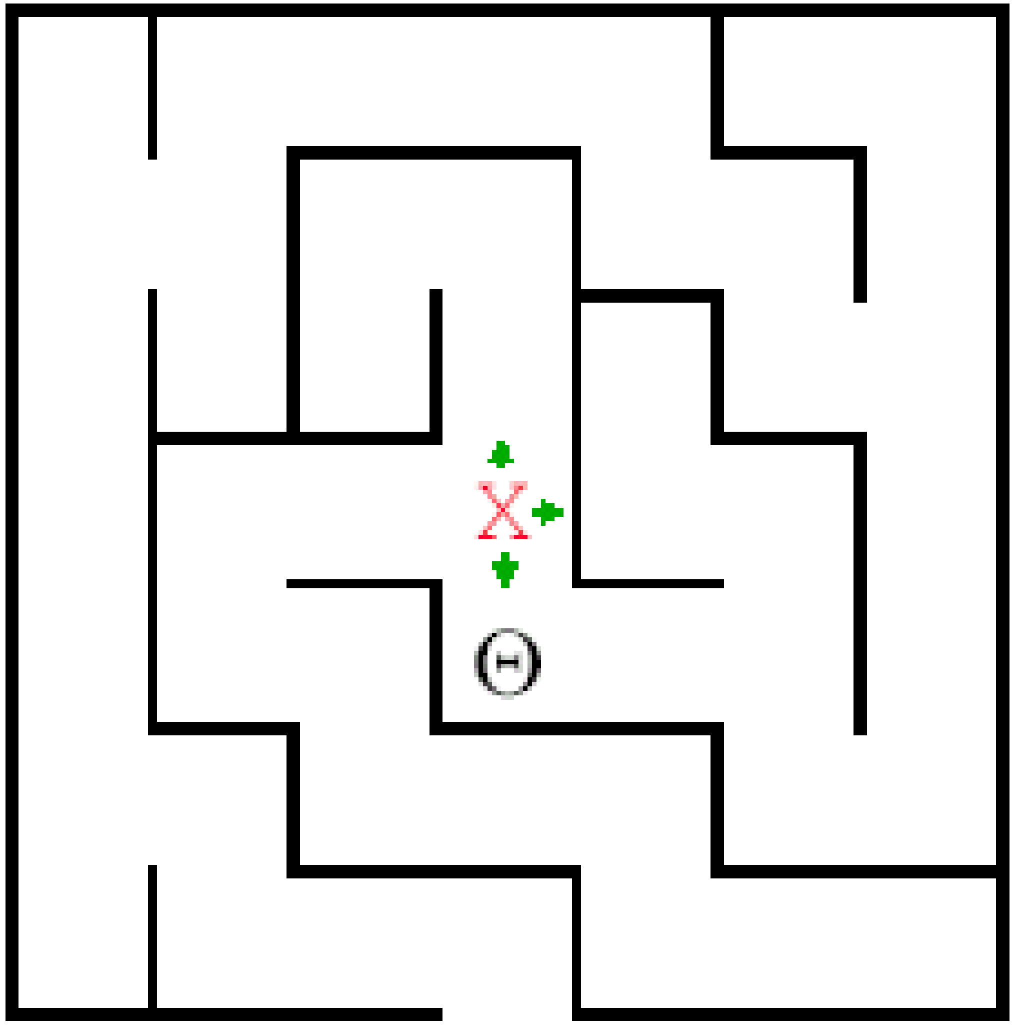
Remember the program stack!

Backtracking in Action



The recursive calls end and return until we find ourselves back here.

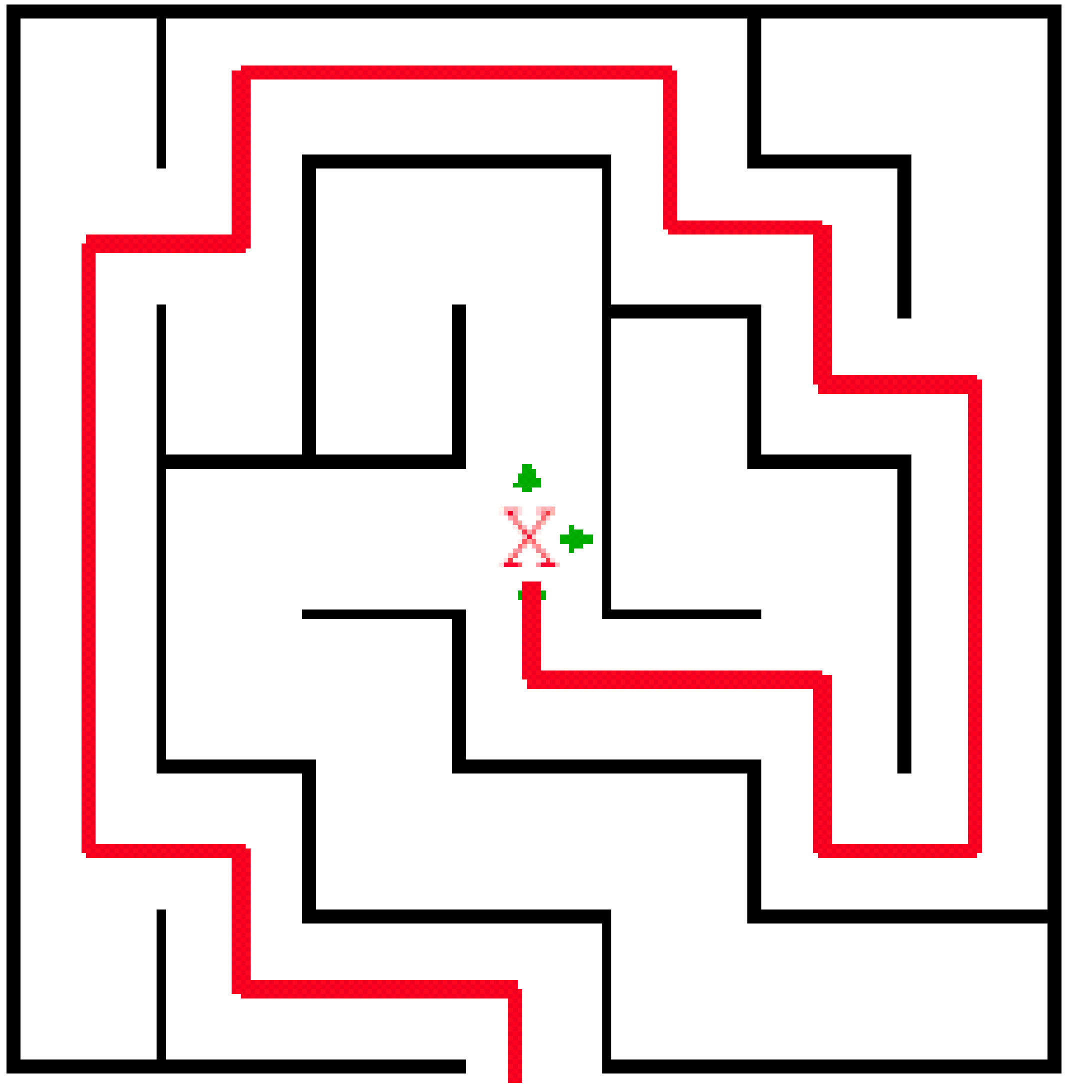
Backtracking in Action



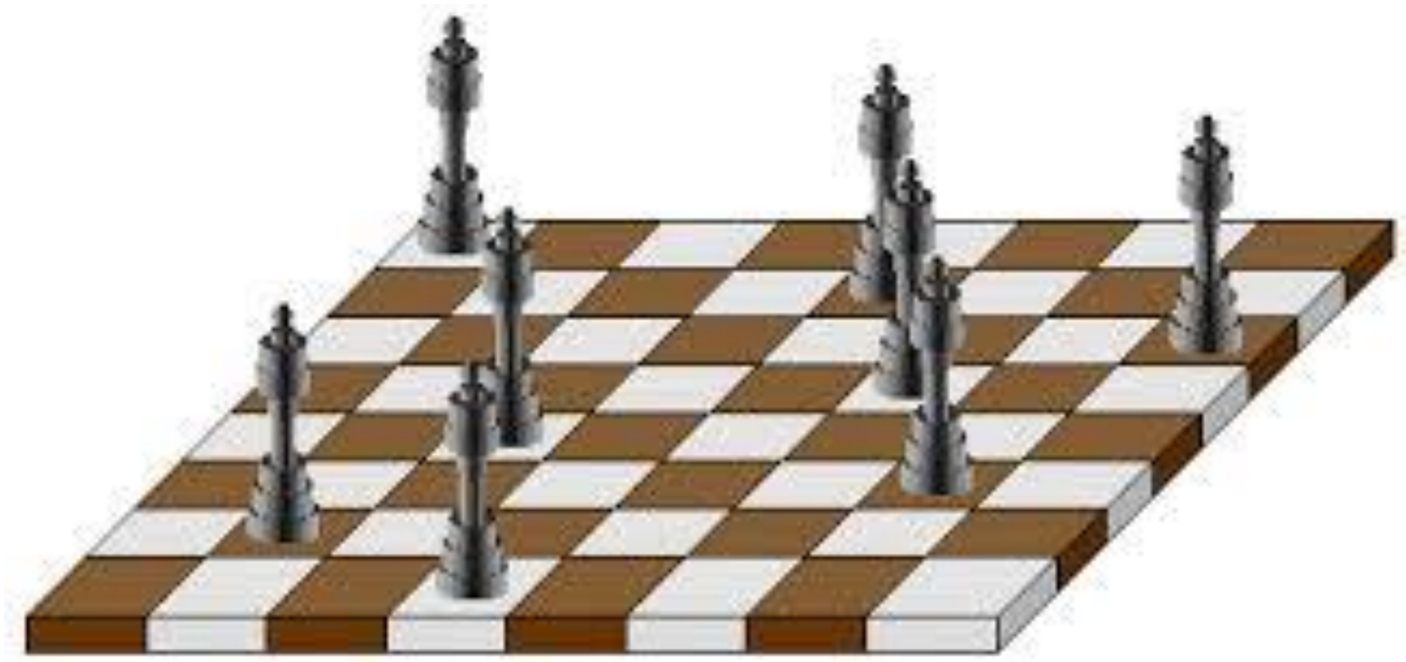
And now we try South

Path Eventually Found

-

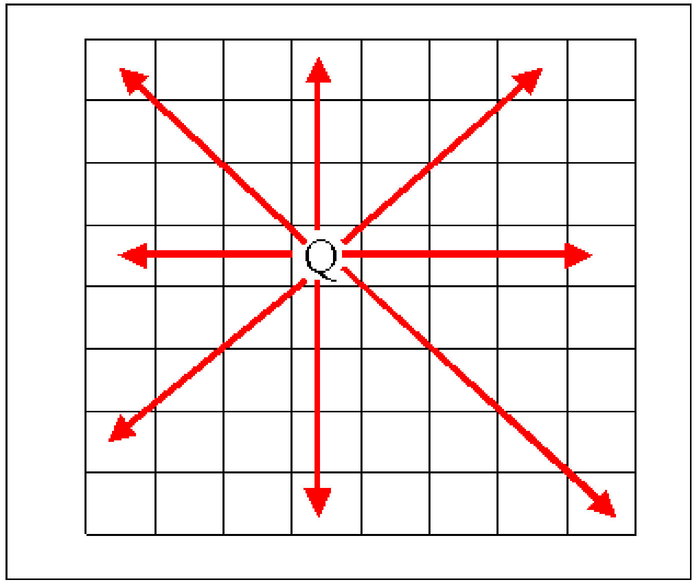


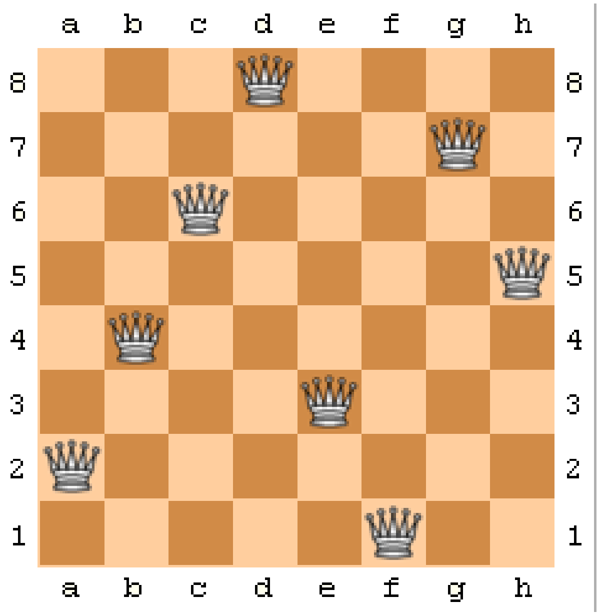
The 8-Queens Problem



The 8-Queens Problem

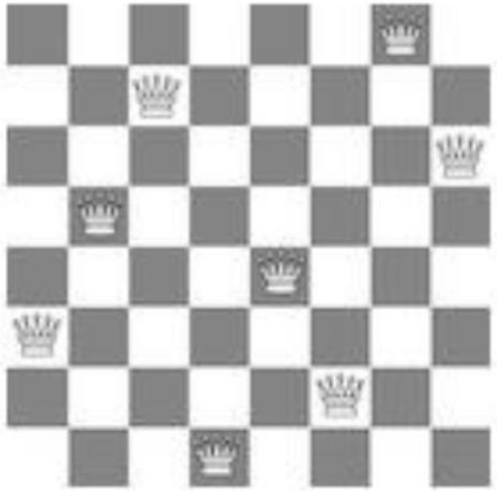
- A classic chess puzzle
  – Place 8 queen pieces on a chess board so that none of them can attack one another

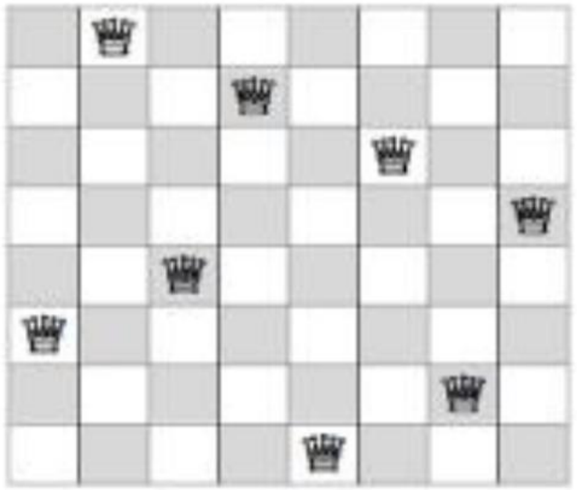




Objective of the Problem

- 8-Queens Problem





- A Queen can move on 8*8 board in horizontally,Verically and diagonally.
- We have to ensure no two Queens should attack each other

Complexity Analysis:
naive (brute force/exhaustive) solution

- One strategy: guess at a solution
  - There are 4,426,165,368 ways to arrange 8 queens on a chessboard of 64 squares

- An observation that eliminates many arrangements from consideration

  - No queen can reside in a row or a column that contains another queen
    - Now: only **40,320 (8!)** arrangements of queens to be checked for attacks along diagonals

Complexity Analysis

- A possible brute-force algorithm for 8-Queen is to generate the permutations of the numbers 1 through 8 (of which there are 8! = **40,320**),
  - and uses the elements of each permutation as indices to place a queen on each row.
  - Then it rejects those boards with diagonal attacking positions.

- The backtracking algorithm, is a slight improvement on the permutation method,
  - constructs the search tree by considering one column (or row) of the board at a time, eliminating most non-solution board positions at a very early stage in their construction.
  - Because it rejects column (or row) and diagonal attacks even on incomplete boards, it examines only **15,720** possible queen placements.

- A further improvement which examines only **5,508** possible queen placements is to combine the permutation-based method with the early pruning method:
  - The permutations are generated depth-first, and the search space is pruned if the partial permutation produces a diagonal attack

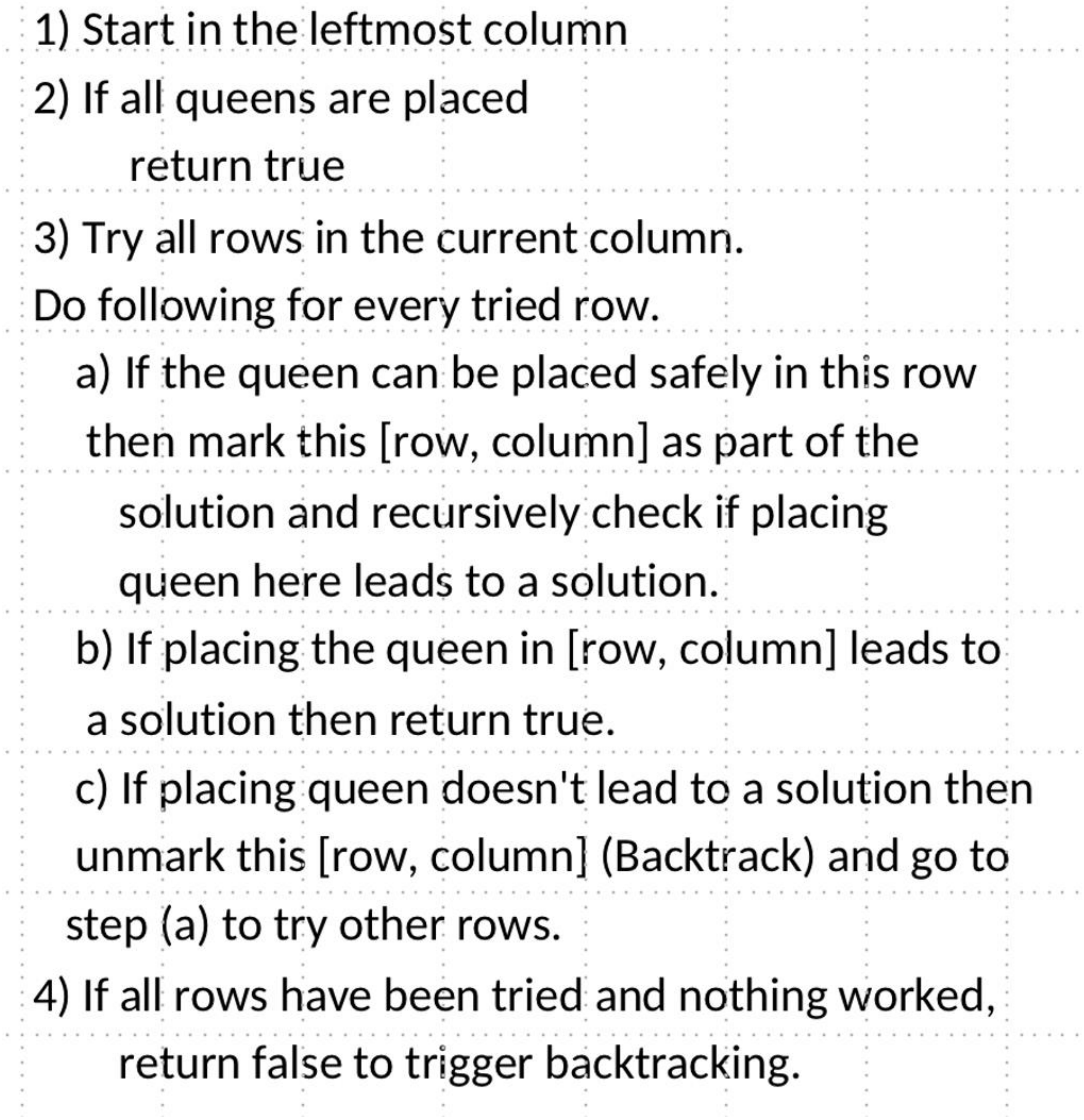
The Eight Queens Problem

- A recursive algorithm that places a queen in a column
    - Base case
        - If there are no more columns to consider
            - You are finished
    - Recursive step
        - If you successfully place a queen in the current column
            - Consider the next column
        - If you cannot place a queen in the current column
            - You need to backtrack

Solve the 8-Queens problem via backtracking

- Backtracking:
  - A systematic way to make successive guesses at a solution.
- If a particular guess leads to a dead end, you back up to that guess and replace it with a different guess.
- This strategy of retracing steps in reverse order and then trying a new sequences of steps is called "backtracking".
- You can combine recursion and backtracking to solve the problem that follows.

The backtracking solution:



```
1) Start in the leftmost column
2) If all queens are placed
       return true
3) Try all rows in the current column.
Do following for every tried row.
   a) If the queen can be placed safely in this row
    then mark this [row, column] as part of the
      solution and recursively check if placing
      queen here leads to a solution.
   b) If placing the queen in [row, column] leads to
    a solution then return true.
   c) If placing queen doesn't lead to a solution then
   unmark this [row, column] (Backtrack) and go to
  step (a) to try other rows.
4) If all rows have been tried and nothing worked,
      return false to trigger backtracking.
```

Search process

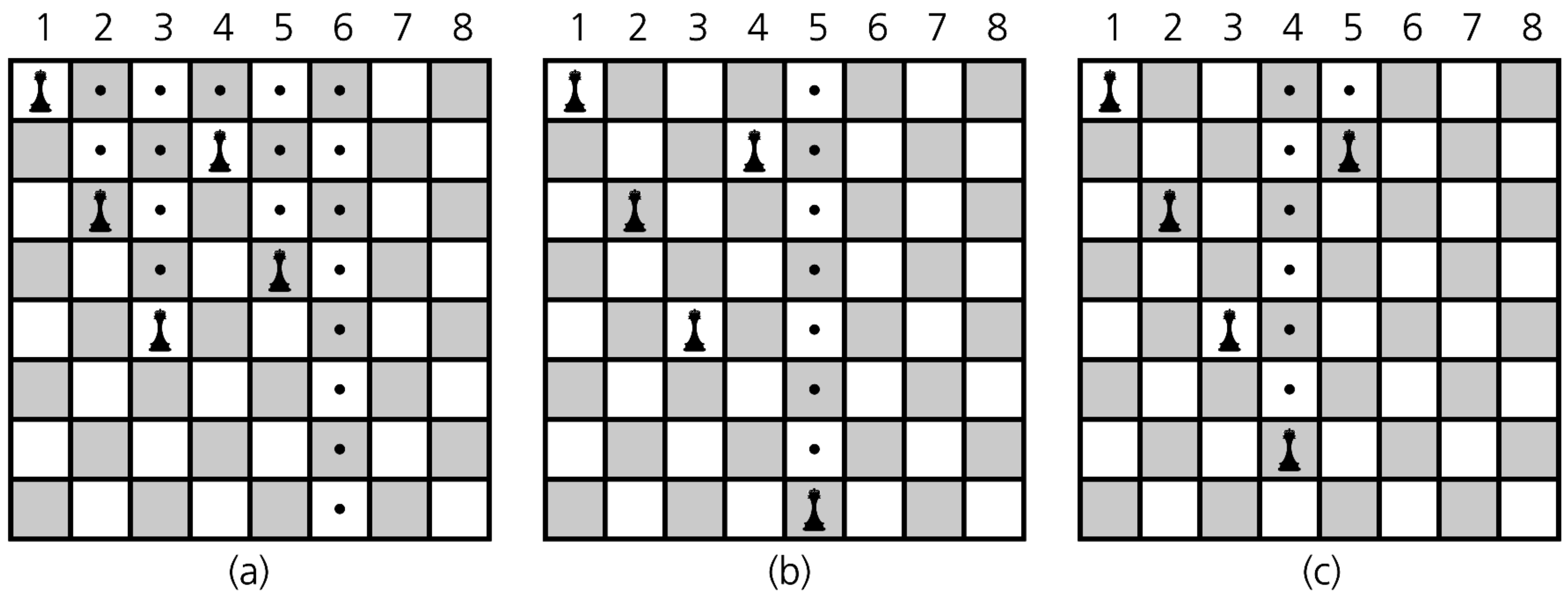


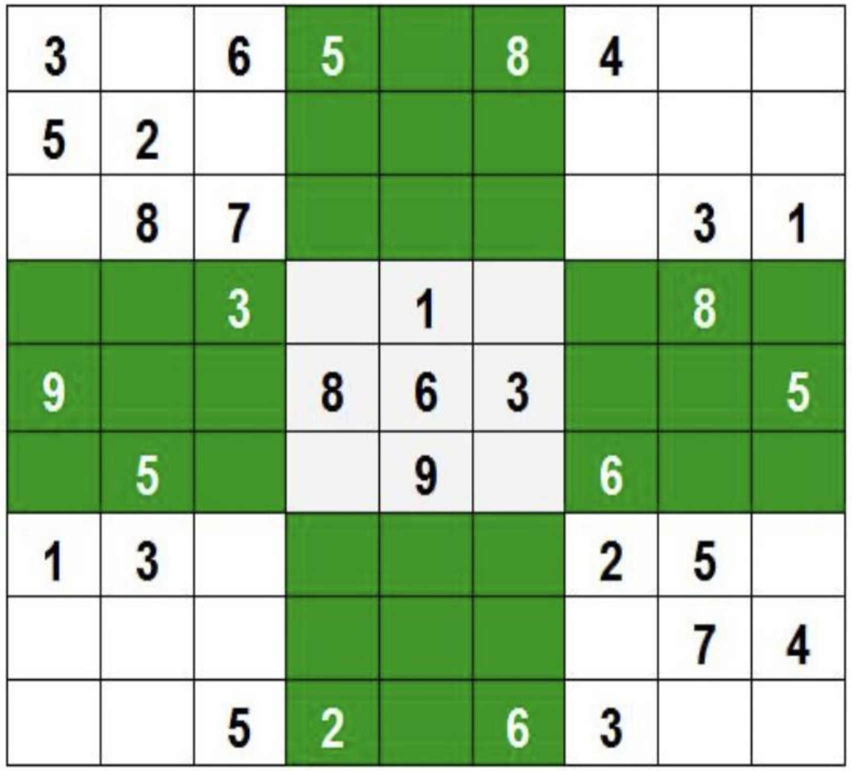
Figure 5-1 (a) Five queens that cannot attack each other, but that can attack all of column 6; (b) backtracking to column 5 to try another square for the queen; (c) backtracking to column 4 to try another square for the queen and then considering column 5 again

Pseudo code:

```
placeEightQueens( chessboard )
  placeQueen ( chessboard,  row = 0 )

placeQueen( chessboard, row )
  if row is greater than 8,
    return true (problem is solved)

  for each column from 0 to 8
    try to add queen to that column,
    - if the row, column position is valid for the new queen
    (i.e., it is not under attack)
    then move on to the next row of the chessboard
    - if placeQueen ( chessboard, row + 1) is successfull
    then return true to stop the for loop from checking
    remaining columns

  return false, no position in the current row is valid
```

Solve Sudoku Puzzle

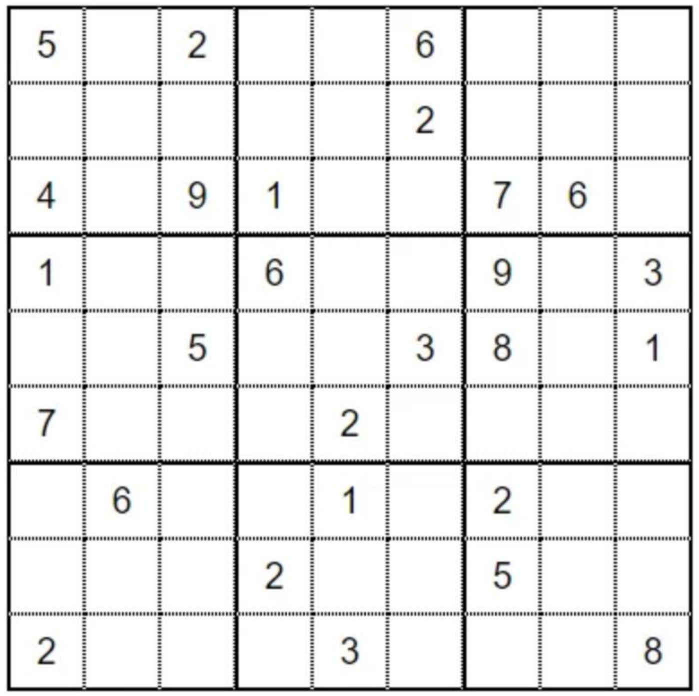
Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.



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

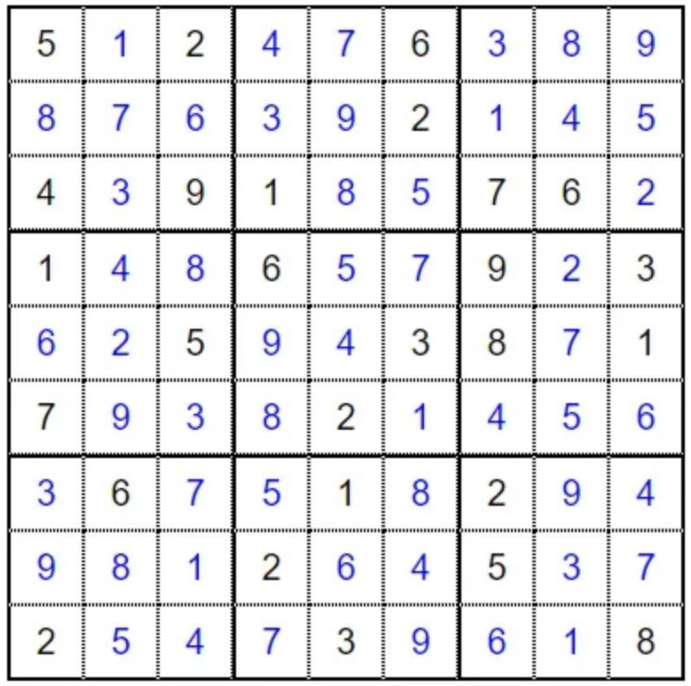
Example

Input :



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

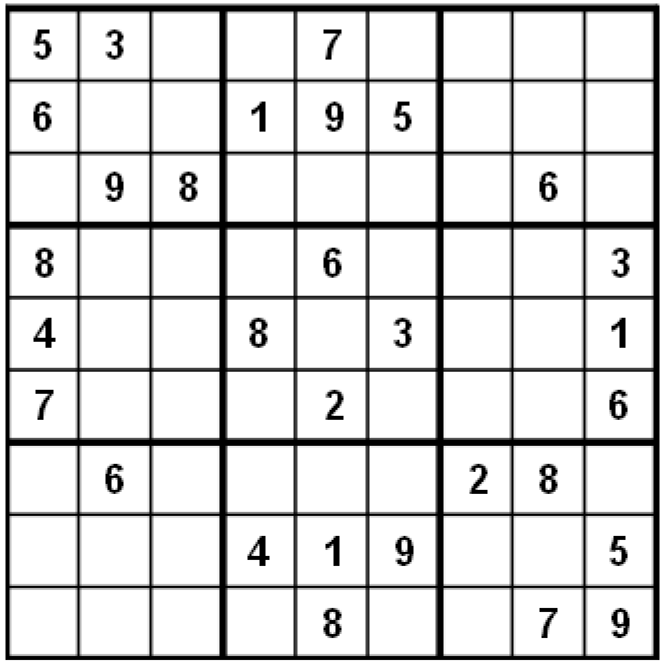
Output :



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

A Concrete Example

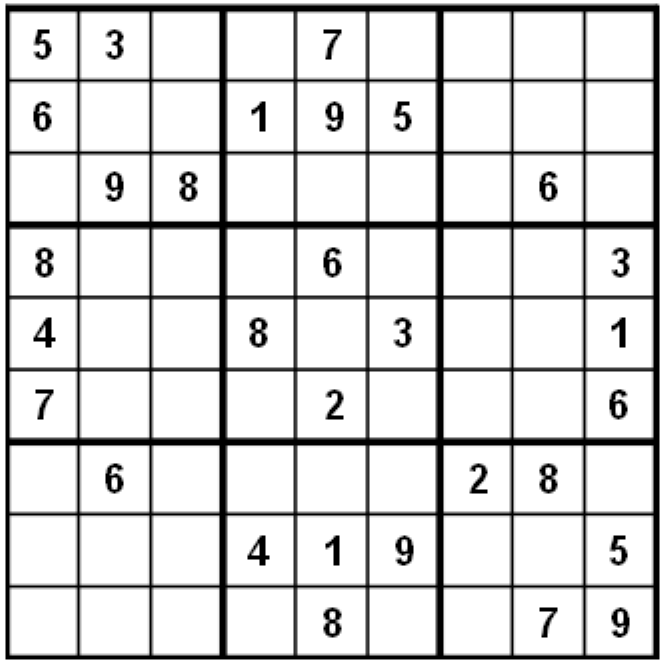
- 9 by 9 matrix with some numbers filled in
- all numbers must be between 1 and 9
- Goal: Each row, each column, and each mini matrix must contain the numbers between 1 and 9 once each
  - no duplicates in rows, columns, or mini matrices



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

Solving Sudoku – Brute Force

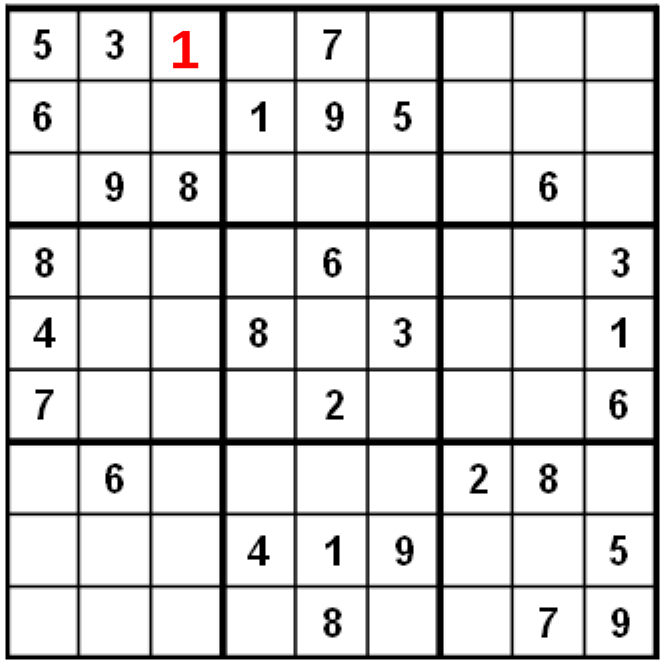
- A *brute force* algorithm is a simple but general approach
- Try all combinations until you find one that works
- This approach isn't clever, but computers are Fast
- If not fast enough, try and improve on the brute force results



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

Solving Sudoku

- Brute force Sudoku Solution
  - if not open cells, solved
  - scan cells from left to right, top to bottom for first open cell
  - When an open cell is found start cycling through digits 1 to 9.
  - When a digit is placed check that the set up is legal
  - now solve the board



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

Solving Sudoku – Later Steps

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

$\rightarrow$

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

$\rightarrow$

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

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

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

Oh no !

Sudoku – A Dead End

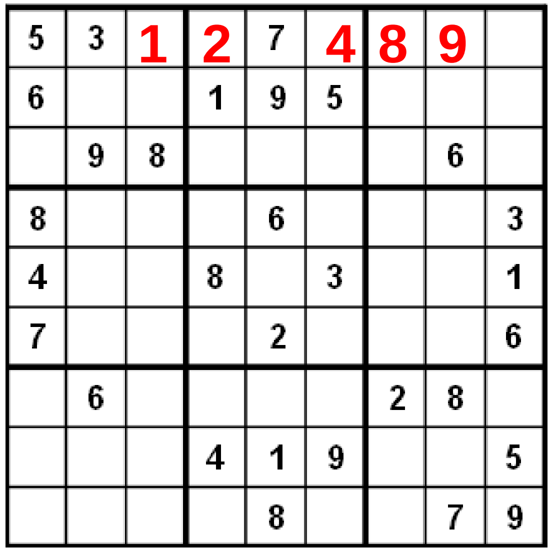
- We have reached a dead end in our search

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

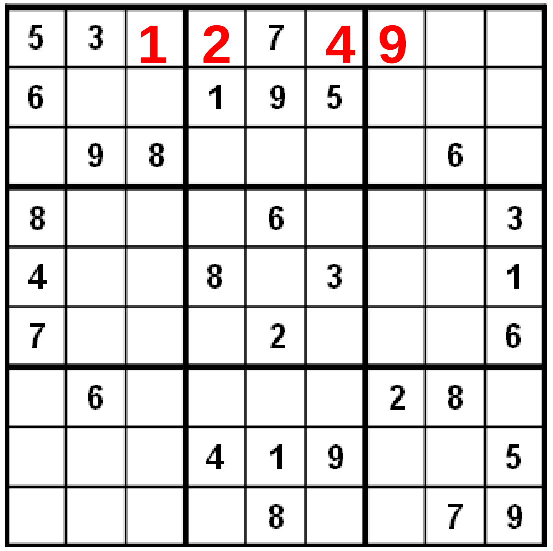
- With the current set up none of the nine digits work in the top right corner

Back track

- When the search reaches a dead end in ***backs up*** to the previous cell it was trying to fill and goes onto to the next digit
- We would back up to the cell with a 9 and that turns out to be a dead end as well so we back up again
  - so the algorithm needs to remember what digit to try next
- Now in the cell with the 8. We try and 9 and move forward again.



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



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

Key Insight

- After trying placing a digit in a cell we want to solve the new sudoku board
  - Isn't that a smaller (or simpler version) of the same problem we started with?
- After placing a number in a cell, we need to remember the next number to try in case things don't work out.
- We need to know if things worked out (found a solution) or they didn't, and if they didn't, try the next number
- If we try all numbers and none of them work in our cell we need to *report back* that things didn't work

Recursive Backtracking

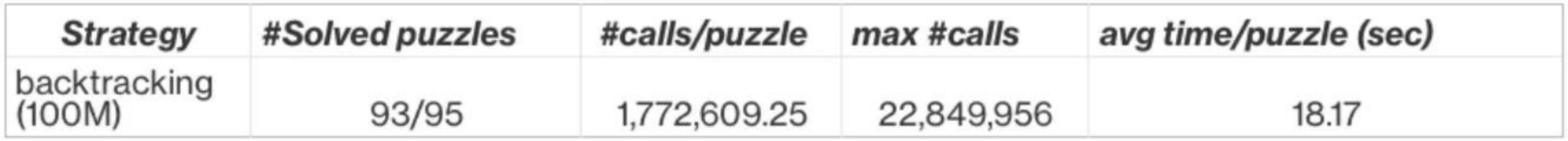
- Problems such as Suduko can be solved using recursive backtracking;
- recursive because later versions of the problem are just slightly simpler versions of the original;
- backtracking because we may have to try different alternatives

Recursive Backtracking Algorithm

Pseudo code for recursive backtracking algorithms:

```
If at a solution, report success
for(every possible choice from current state/node)
  Make that choice and take one step along path
  Use recursion to solve the problem for the new node/state
  If the recursive call succeeds, report the success to the
  next high level
  Back out of the current choice to restore the state at the
  beginning of the loop.
Report failure
```

Timing of the above algorithm on a standard laptop



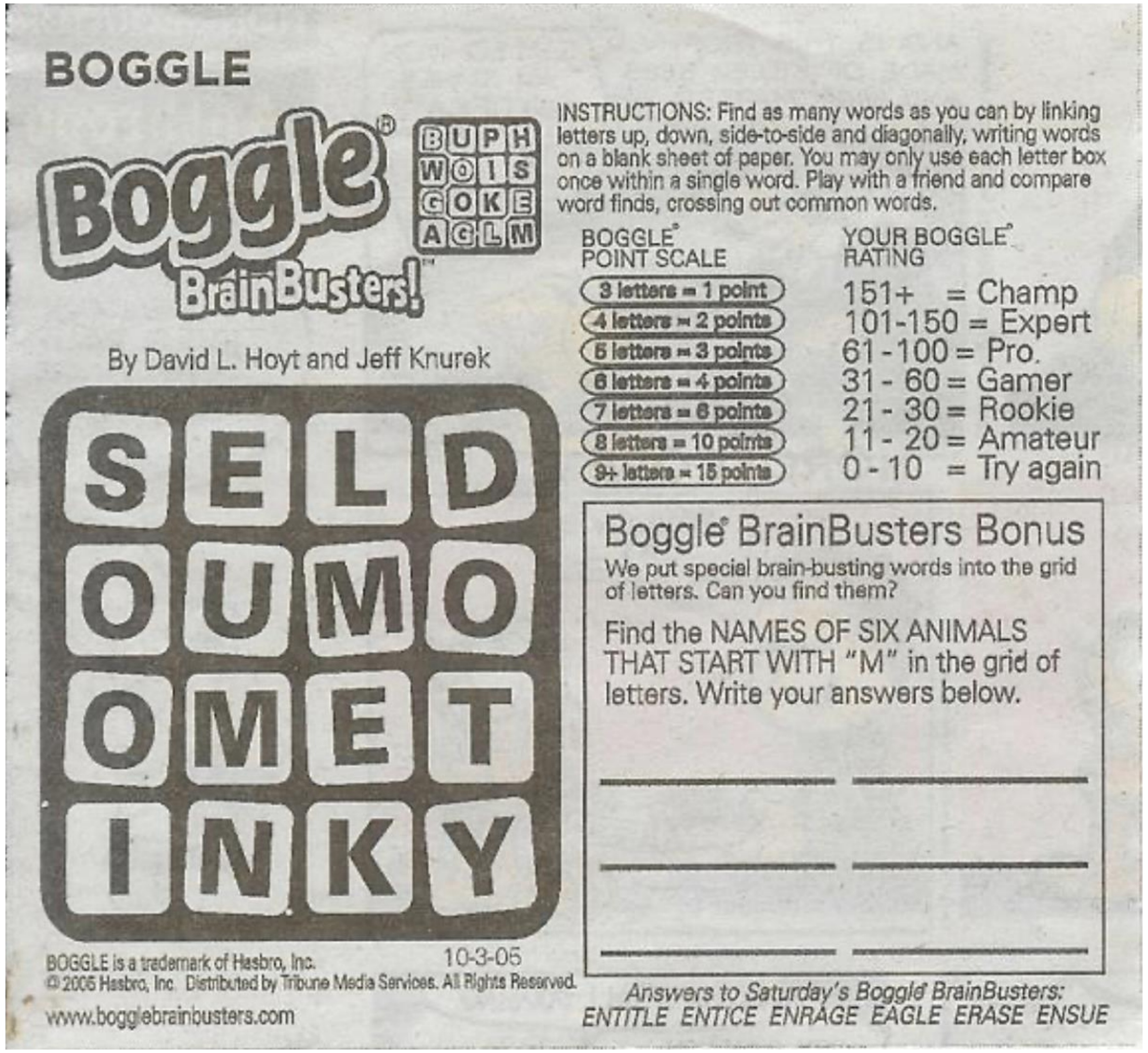
Strategy	#Solved puzzles	#calls/puzzle	max #calls	avg time/puzzle (sec)
backtracking (100M)	93/95	1,772,609.25	22,849,956	18.17

Table 1: Search results of Backtracking algorithm when limited to 100M calls.

A challenge: Can you develop a new algorithm that is by ~ 1,000 times fasters (on the same laptop) ?

(around 20 ms/puzzle)

Boggle words search



BOGGLE

Boggle® BrainBusters!™

By David L. Hoyt and Jeff Knurek

S	E	L	D
O	U	M	O
O	M	E	T
I	N	K	Y

INSTRUCTIONS: Find as many words as you can by linking letters up, down, side-to-side and diagonally, writing words on a blank sheet of paper. You may only use each letter box once within a single word. Play with a friend and compare word finds, crossing out common words.

BOGGLE® POINT SCALE

- 3 letters = 1 point
- 4 letters = 2 points
- 5 letters = 3 points
- 6 letters = 4 points
- 7 letters = 6 points
- 8 letters = 10 points
- 9+ letters = 15 points

YOUR BOGGLE® RATING

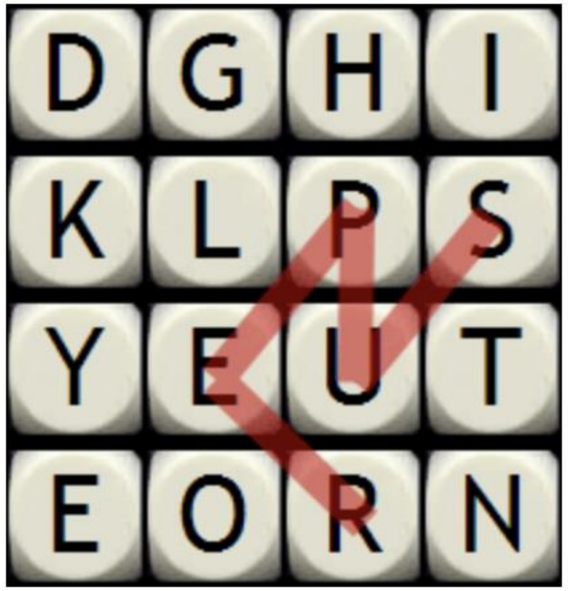
151+	= Champ
101-150	= Expert
61 - 100	= Pro.
31 - 60	= Gamer
21 - 30	= Rookie
11 - 20	= Amateur
0 - 10	= Try again

Boggle® BrainBusters Bonus

We put special brain-busting words into the grid of letters. Can you find them?

Find the NAMES OF SIX ANIMALS THAT START WITH "M" in the grid of letters. Write your answers below.

Answers to Saturday's Boggle® BrainBusters:
ENTITLE ENTICE ENRAGE EAGLE ERASE ENSUE



Backtracking Solution

We'll try each tile as the beginning of our traversal, and what we'll recursively do is:

1. Check if the tile is valid. (In-bounds in the board ,and matches the character in the word.)
2. Invalidate that tile by changing it to an invalid character, like "*". (Which we know will never be in our input.)
3. Recursively check the left, right, up, and down tiles.
4. Undo the tile invalidation from step 2.
5. Return the true if this tile was the end of the word or if any of the recursive calls returned true.

The "backtracking" part of this solution is that we're changing shared state (step 2), making recursive calls, and then undoing our change for when we go back up to previous levels.

Summary of today’s lecture

- 1. The recursive way of thinking
    - Prove the correctness of recursion
    - Recursion versus iteration
- 2. Recursion as a problem solving technique
    - Solve the Towers of Hanoi puzzle
- 3. Recursive backtracking
    - Examples:
        - Maze puzzle
        - Eight-Queens puzzle
        - Sudoku puzzle
        - …

Take-home exercises: (for self test, not to be assessed)

1. write a recursive java method that writes the digits of a positive decimal integer in reverse order.

2. Write a recursive method that computes the product of all the items in the array A[first.. last].

3. Consider a 4-Queens problem, which has the same rules as the 8-Queens problem but uses a 4x4 chessboard. Find all solutions to this new problem by applying backtracking by hand.

- (continue on the next page...)

4. Consider the following recursive method:

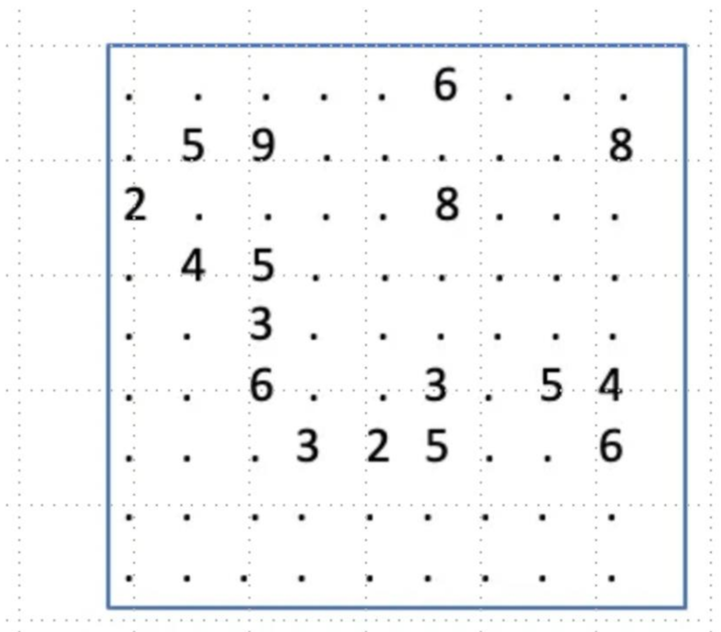
```
 public static int p (int x){
  if (x<3) {
        return x;
  }
 else {
       return p(x-1) * p(x-3) ;
        } //end if
  } //end p
```

Let m(x) be the number of multiplications that the execution of p(x) performs.

a. Write recursive definition of m(x).
b. Prove that your answer to Part a is correct by using mathematical induction.

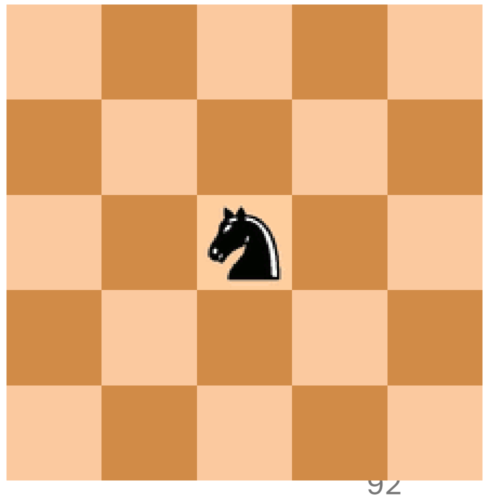
5. Solve the Sudoku puzzle as shown in the figure in the right, using the Java recursive algorithm.

6. Solve the one knight tour puzzle on the 8x8 chessboard using backtracking.
(https://en.wikipedia.org/wiki/Knight%27s_tour)



```
. . . . . 6 . . .
. 5 9 . . . . . 8
2 . . . . 8 . . .
. 4 5 . . . . . .
. . 3 . . . . . .
. . 6 . . 3 . 5 4
. . . 3 2 5 . . 6
. . . . . . . . .
. . . . . . . . .
```

Figure 4: A killer puzzle generated by Norvig



End of Lecture 7A