Survey of Scientific Computing

(SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 15

Combinatorics, Encoding, Search

Session Goals

- Trace how **recursion** can be used to calculate factorials
- Learn the rules for the **Scramble Squares** puzzle
- Formulate **research questions** to analyze the puzzle
- Develop a taxonomy to differentiate puzzle images
- Consider how to **encode** the puzzle pieces
- Develop a method to describe a puzzle **solution**
- Understand the **algorithm** to check for a valid layout
- Appreciate how **recursion** can be used to solve the puzzle

Recursion

- Recursion is when a function calls **itself**
- Each invocation (instance) of the function has its own copy of its **local variables**
- At the end of the function, the program **returns** to the place in the source code where it was previously called
- Return values can be passed all the way back up the **call stack** to the very first calling function
- A **terminating condition** must exist to avoid an infinite loop and stack crash

Iterative Factorial

In mathematics, the **factorial** of a non-negative integer *n*, denoted by *n*!, is the product of all positive integers less than or equal to *n*. For example,

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

The value of 0! is 1, according to the convention for an empty product.[1]

```
int main()
{
    int x = 5;

    cout << "Iterative Factorial of "
         << x << "! = "
         << IterativeFactorial(x) << endl;

    return 0;
}
```

```
int IterativeFactorial(int n)
{
    int x = n;
    while (n > 1) {
        n = n-1;
        x = x * n;
    }
    return x;
}
```

```
File   Edit   View   Terminal   Tabs   Help
Iterative Factorial of 5! = 120

Process returned 0 (0x0)   executi
Press ENTER to continue.
```

Recursive Factorial Call Stack

RecursiveFactorial(5)						returns 120					
↳	RecursiveFactorial(4)						returns 24				
	↳	RecursiveFactorial(3)						returns 6			
		↳	RecursiveFactorial(2)						returns 2		
			↳	RecursiveFactorial(1)						returns 1	
				↳	RecursiveFactorial(0)	→					returns 1

```
int IterativeFactorial(int n)
{
    int x = n;
    while (n > 1) {
        n = n-1;
        x = x * n;
    }
    return x;
}
```

```
int RecursiveFactorial(int n)
{
    if (n==0)
        return 1;
    else
        return n * RecursiveFactorial(n-1);
}
```

Recursive code is often a more “natural” expression of the problem

Run Lab 1 – Factorial Recursive

$$N! = N * (N-1)!$$

where 0! = 1

```
int RecursiveFactorial(int n)
{
    if (n==0)
        return 1;
    else
        return n * RecursiveFactorial(n-1);
}
```

Terminating Condition

```
int main()
{
    int x = 5;

    cout << "Iterative Factorial of "
         << x << "! = "
         << IterativeFactorial(x) << endl;

    cout << "Recursive Factorial of "
         << x << "! = "
         << RecursiveFactorial(x) << endl;

    return 0;
}
```

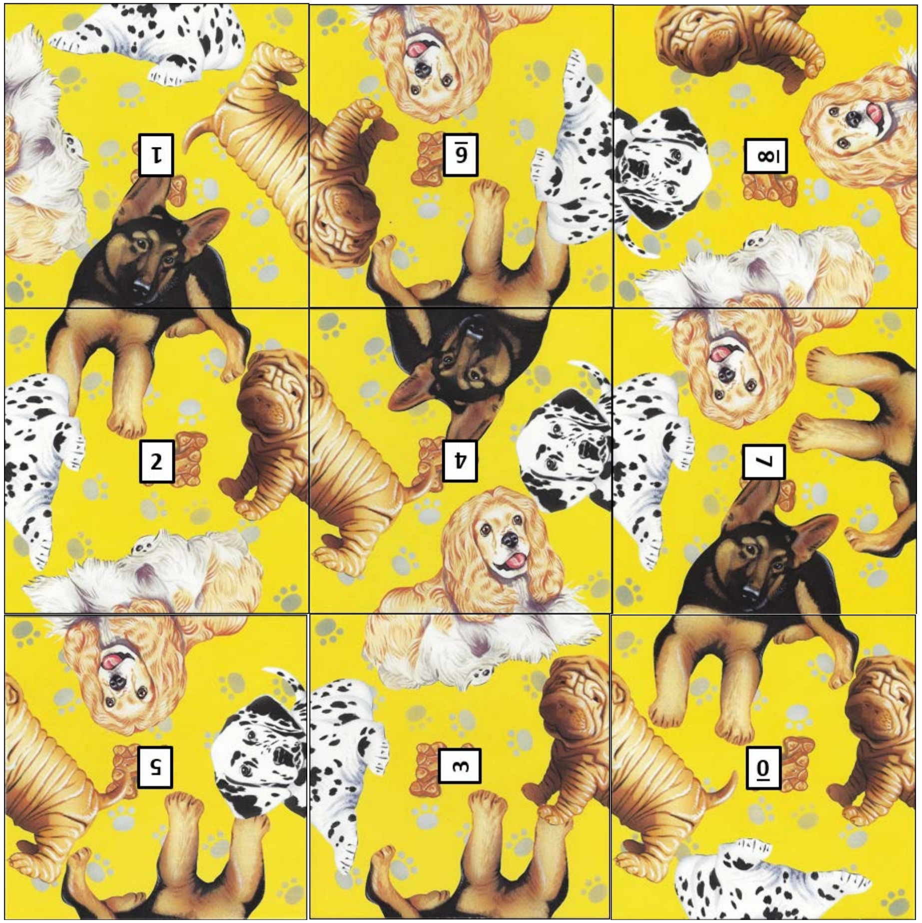
File Edit View Terminal Tabs Help

```
Iterative Factorial of 5! = 120
Recursive Factorial of 5! = 120

Process returned 0 (0x0)   executi
Press ENTER to continue.
```

Scramble Squares

- There are **9** tiles in a **3 x 3** matrix
- Each tile can be rotated in **4** different positions
- Inside edges of **adjacent** tiles must make a full image (be complimentary)
- Edges around outside of entire matrix don’t need to match the other side
- There may be multiple solutions (layouts) but **there is always at least one**



Frame of Reference

- 60 seconds in one minute
- 60 minutes in one hour
- 24 hours in one day
- 365 days in one year
- 60 x 60 x 24 x 365 = **~ 32M**

- Remember this frame of reference:

There are <u>only</u> 32 million seconds in a year

Frame of Reference

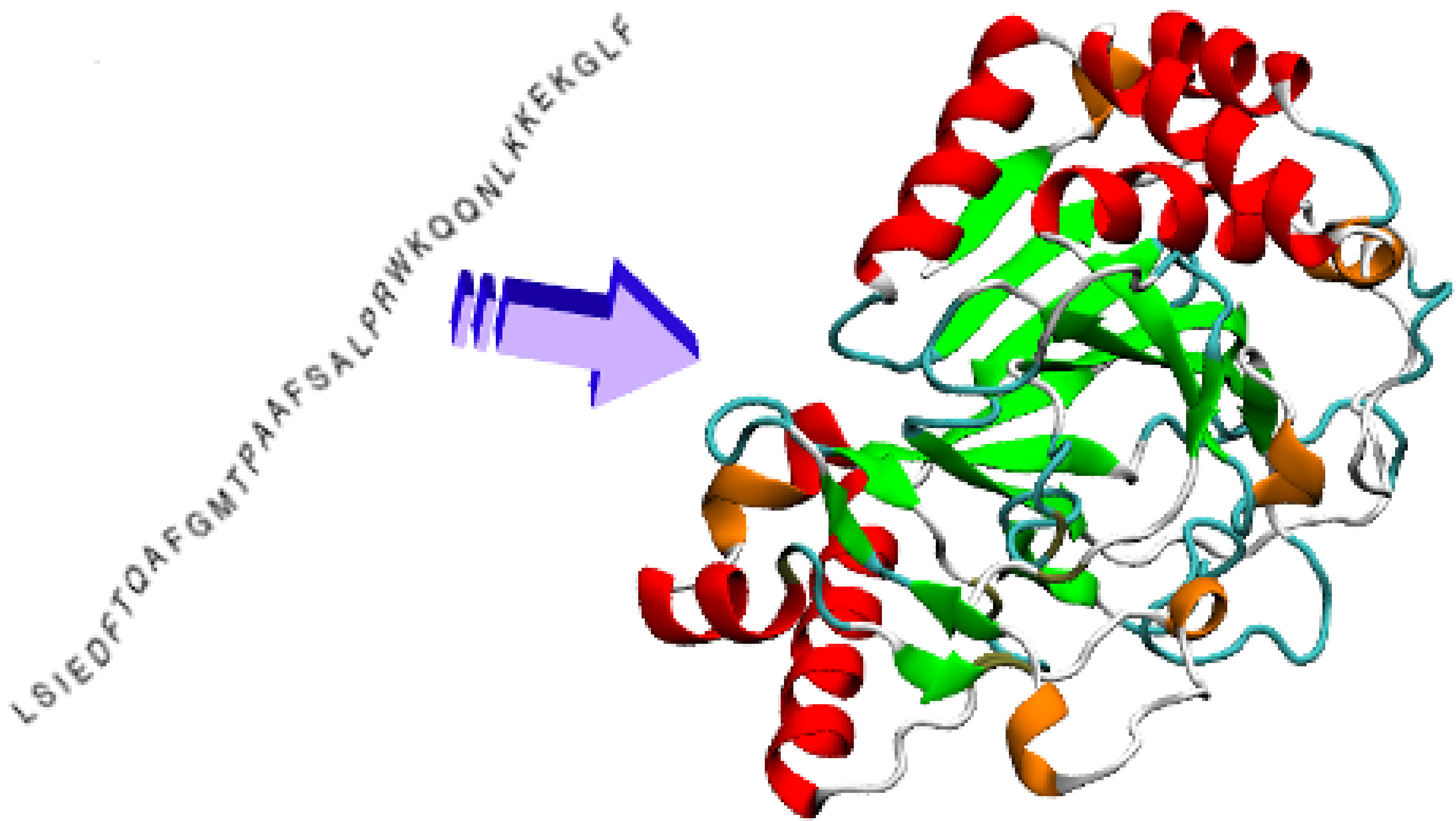
- 9 scramble squares
- 4 possible rotations per square
- 9 x 4 = 36 possible "squares"
- 36 permute 9 = ~
- Even if

  to try them all !!
- Ren erence:

  are only 32 million seconds in a year

We cannot solve scramble squares using brute force methods!

Why study Scramble Squares?



Why study Scramble Squares?

Scramble Squares

- 4 image types
- Tiles have binding sit…
- Tiles rotate th… fixed l…
- Ac… hav… ima…
- Avoi… possible layouts

Protein Struct…

- 4 b… NA
- …ding sites
- …ngles
- …ophobic and hydrophilic sections must align
- Avoid high energy conformations

We cannot solve the protein folding problem using brute force methods!

First Things First

- Open the bags carefully – they rip easily
- **Never write anything directly on a puzzle piece!**
- Verify you have **9 puzzle pieces**
- Each puzzle has a **two letter identifier** on the **back** (bottom right corner) of every piece
- Ensure all 9 pieces in your puzzle **have the same two-letter identifier**
- Lay all the pieces (**picture side up**) in a **3 x 3 grid**

Solve It Like A Scientist!

- Don't just look at the pieces – ***see*** them
- What **research questions** can you pose about your puzzle set?
- What types of **analysis *of each piece*** could help guide you to a solution?
- How can we **encode** the order and orientation of each puzzle piece, so you can tell me your solution ***over the phone?***

Research Questions

1. Which tile has the most number of **head** images?

Tile	0	1	2	3	4	5	6	7	8
Dalmatian Head					1	1			1
Shar-Pei Head	1		1	1			1		1
Spaniel Head					1	1	1	1	1
Shepherd Head		1			1			1	
Totals	1	1	1	1	3	2	2	2	3

Research Questions

2. Which tile has the most number of **body** images?

Tile	0	1	2	3	4	5	6	7	8
Dalmatian Body	1	1	1	1			1	1	
Shar-Pei Body	1	1			1	1			
Spaniel Body		1	1	1					1
Shepherd Body	1		1	1		1	1	1	
Totals	3	3	3	3	1	2	2	2	1

Research Questions

3. Which breed has the largest ***disparity*** (difference) between the numbers of head images vs. body images throughout your entire deck?

Tile	0	1	2	3	4	5	6	7	8	Total
Dalmatian Head					1	1			1	3
Dalmatian Body	1	1	1	1			1	1		6
Shar-Pei Head	1		1	1			1		1	5
Shar-Pei Body	1	1			1	1				4
Spaniel Head					1	1	1	1	1	5
Spaniel Body		1	1	1					1	4
Shepherd Head		1			1			1		3
Shepherd Body	1		1	1		1	1	1		6

Solve It Like A Scientist!

- Now try to solve your **Scramble Square** puzzle!
- Let the **statistics** of your puzzle guide you
  - What position is hardest to place?
  - What breed is hardest to match?
- Good scientists keep good notebooks – write down **what you are doing, why, and what happened**
  - Luck is not reproducible! **What is your *process*?**
  - Develop a **taxonomy** for identifying puzzle images

Identify Your Half Images

- Create a table with **8 rows x 2 columns**
  - **4** species x **2** rows per species (one row for each distinct half image)
  - **Two** columns: one for the half image **label**, the other for the **unique value #** for that half image
- Suggested possible labels:
  - Type **# 1**: Top, Head, Front, Bow, Eyes, Mouth, Beak, Flowers, Northern Hemisphere, etc.
  - Type **# 2**: Bottom, Body, Back, Stern, Tail, Feet, Talons, Bumper, Stem, Base, Basket, Southern Hemisphere, etc.

Identify Your Half Images

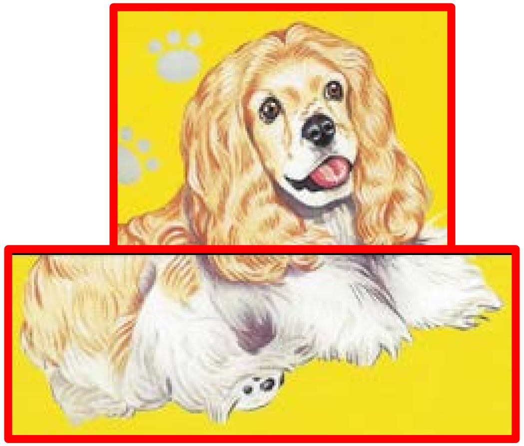
Half Image Identifier	**Value**
Your Species # **1** - Half Image # **1** Identifier	1
Your Species # **1** - Half Image # **2** Identifier	2
Your Species # **2** - Half Image # **1** Identifier	3
Your Species # **2** - Half Image # **2** Identifier	4
Your Species # **3** - Half Image # **1** Identifier	5
Your Species # **3** - Half Image # **2** Identifier	6
Your Species # **4** - Half Image # **1** Identifier	7
Your Species # **4** - Half Image # **2** Identifier	8

Identify Your Half Images

Dalmatian

Shar-Pei

Spaniel



Shephard

Identify Your Half Images

Half Image Identifier	Value
Dalmatian Head	1
Dalmatian Body	2
Shar-Pei Head	3
Shar-Pei Body	4
Spaniel Head	5
Spaniel Body	6
Shephard Head	7
Shephard Body	8

Encoding Half Images (Base 10)

Complimentary **Valid** Sums?


(1) + 
(2) **= 3**


(3) + 
(4) **= 7**

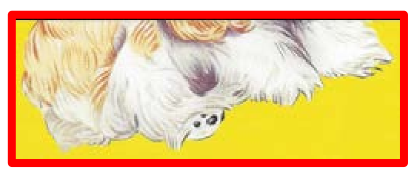

(5) + 
(6) **= 11**


(7) + 
(8) **= 15**

The Problem With Base 10

 (3) +  (4) = 7

 (5) +  (2) = 7 Invalid match!

 (1) +  (6) = 7 Invalid match!

Encoding Half Images (Powers of 2)

Complimentary **Valid** Sums!

 (1) +  (2) = 3

 (4) +  (8) = 12

 (16) +  (32) = 48

 (64) +  (128) = 192

Only valid matches can produce these four sums!

Encoding Half Images (Powers of 2)

 (1) +  (32) = 31 Invalid sum!

 (4) +  (8) = 12 

 (4) +  (1) = 5 Invalid sum!

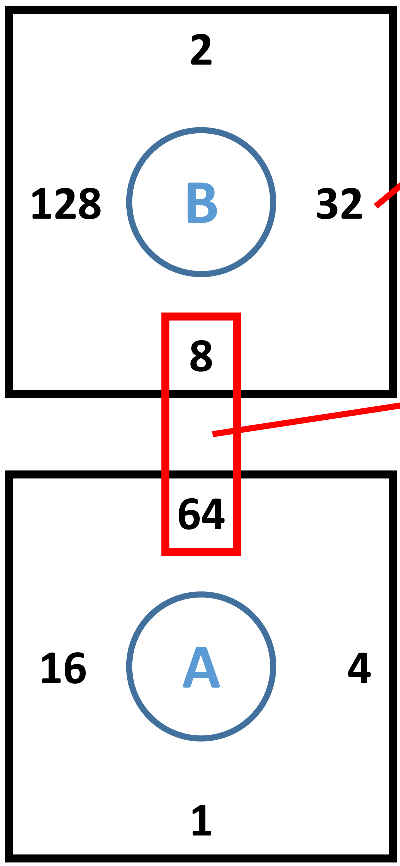
By encoding each half image with an increasing power of 2,
valid complimentary image matches will have a **sum** of ***ONLY*** **3, 12, 48, or 192** between adjacent binding sites.

Identify Your Half Images

Half Image Identifier	Value
Dalmatian Head	1
Dalmatian Body	2
Shar-Pei Head	4
Shar-Pei Body	8
Spaniel Head	16
Spaniel Body	32
Shephard Head	64
Shephard Body	128

Change your numbers to increase by **powers of 2**

Complimentary Half Images



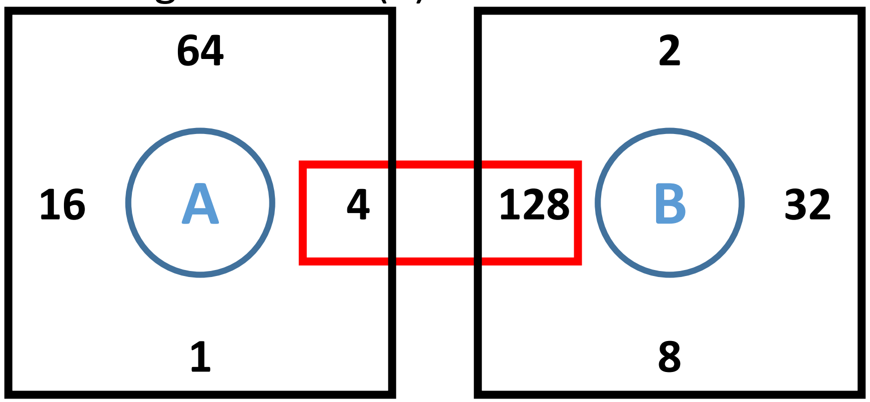
These are the half image power of 2 values stored in each binding site, for the current rotation of tiles A & B

Sum the NORTH binding (half image value) of Tile A and the SOUTH binding (half image value) of Tile B

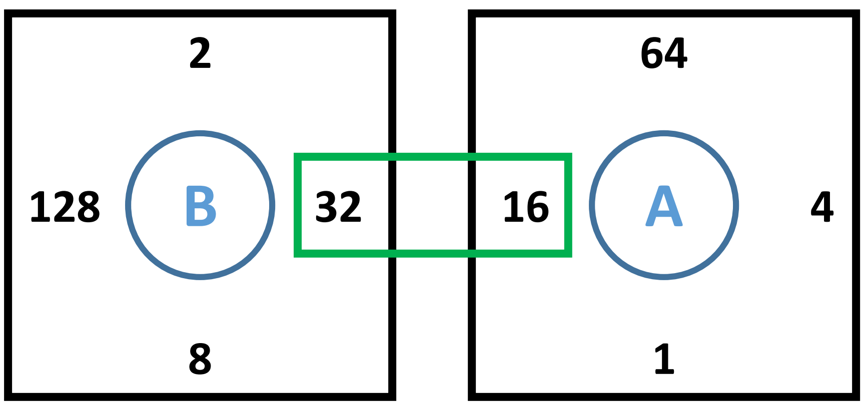
Checking the NORTH (0) binding site:
Tile A North binding = 64
Tile B South binding = 8
Since the sum (64 + 8 = 72) is **not** one of the allowed values {3, 12, 48, or 192},
Tile A **cannot** be placed in this position!

Magic Numbers:
3, 12, 48, or 192

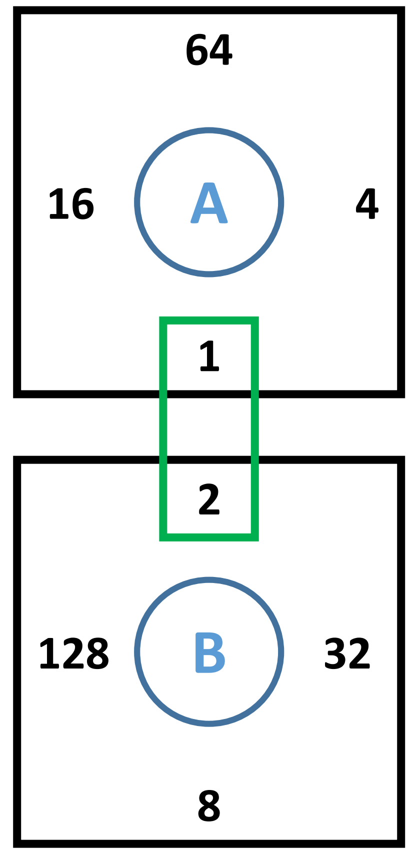
Checking EAST site (1) **sum = 132** : **No Good!**



Checking WEST site (3) **sum = 48** : **Good!**



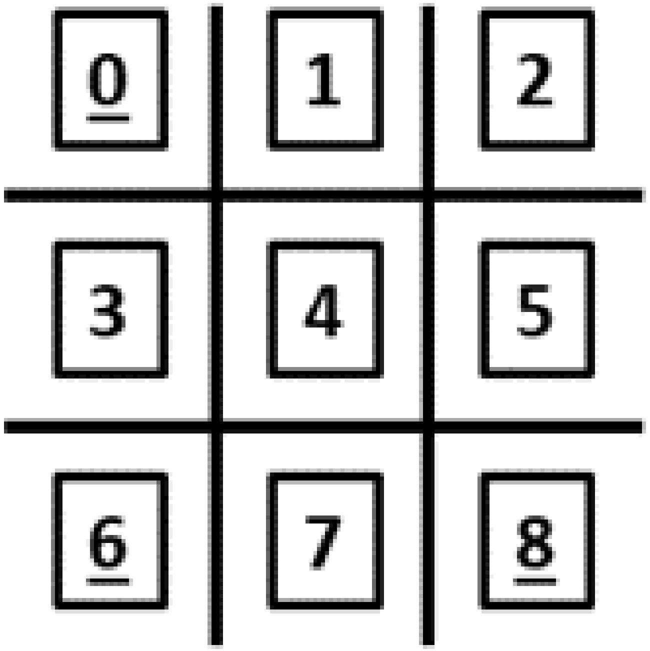
Checking SOUTH site (2)
sum = 3 : **Good!**



Encoding Tile **Id** and Position **#**

- Every tile has an Id (**0 – 8)**
- Every position has a **#** (**0 – 8)**

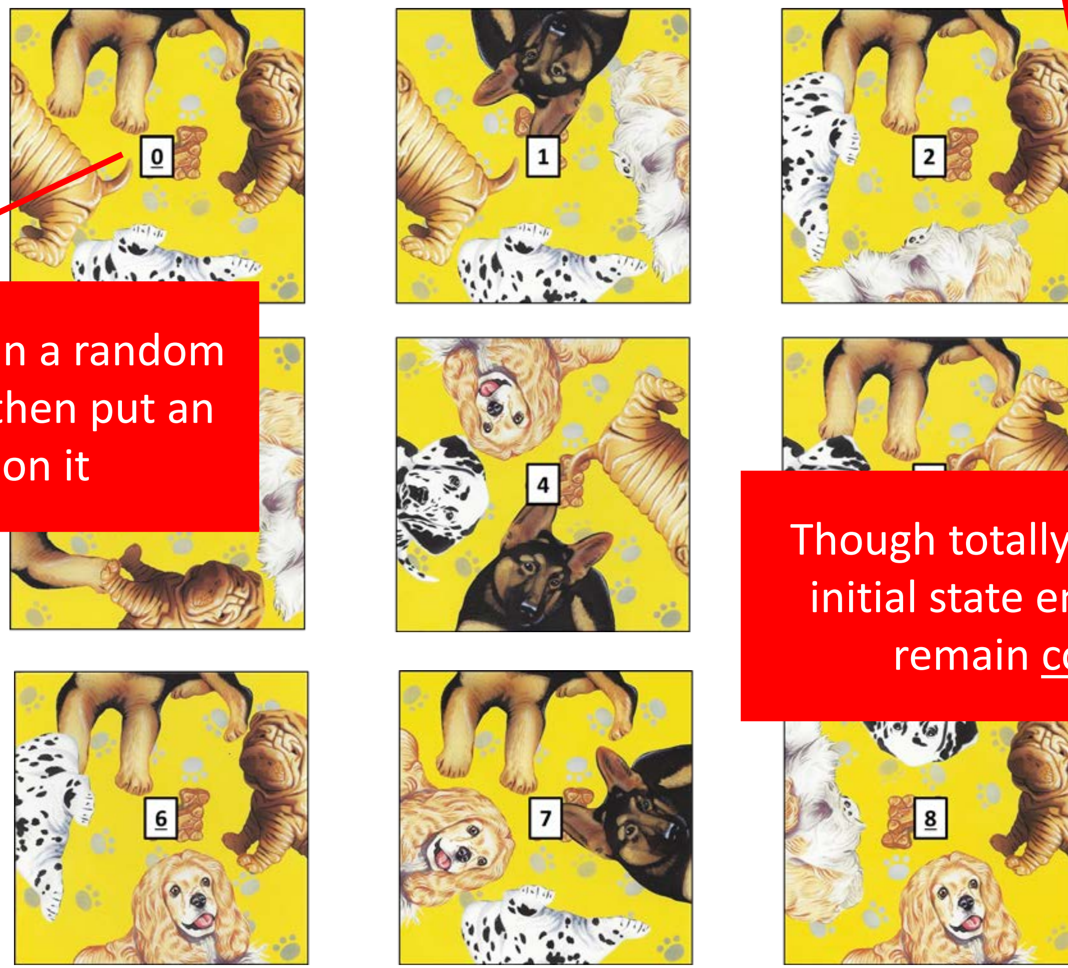
Position Numbers



Preparing Your Scramble Squares

- Place a **Post-It** note on the **center front** of *each* of your tiles
  - Gently remove any existing post-it notes
  - Please **do not** write on the puzzle pieces themselves!
- Write a single number **(0-8)** in the **middle** of each Post-It note to designate the **Tile Id #**
  - Be sure to <u>underline</u> each Tile Id # to clearly designate the original orientation of each tile
  - The underline will help you remember which half image was in the North position for each Tile Id #

Initial State Assignment is Arbitrary

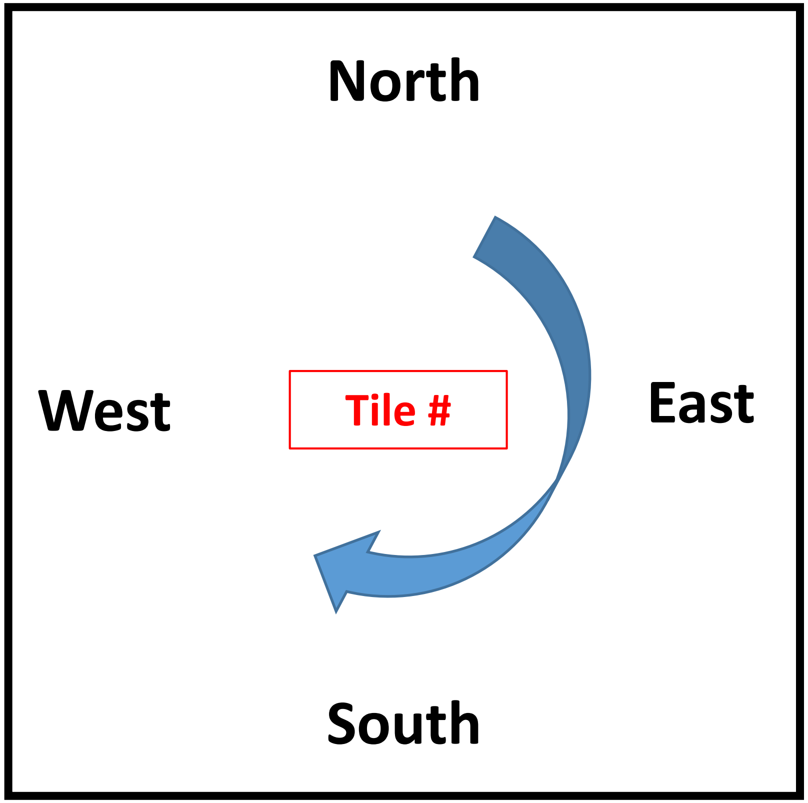


I placed each tile in a random orientation, and then put an Id number on it

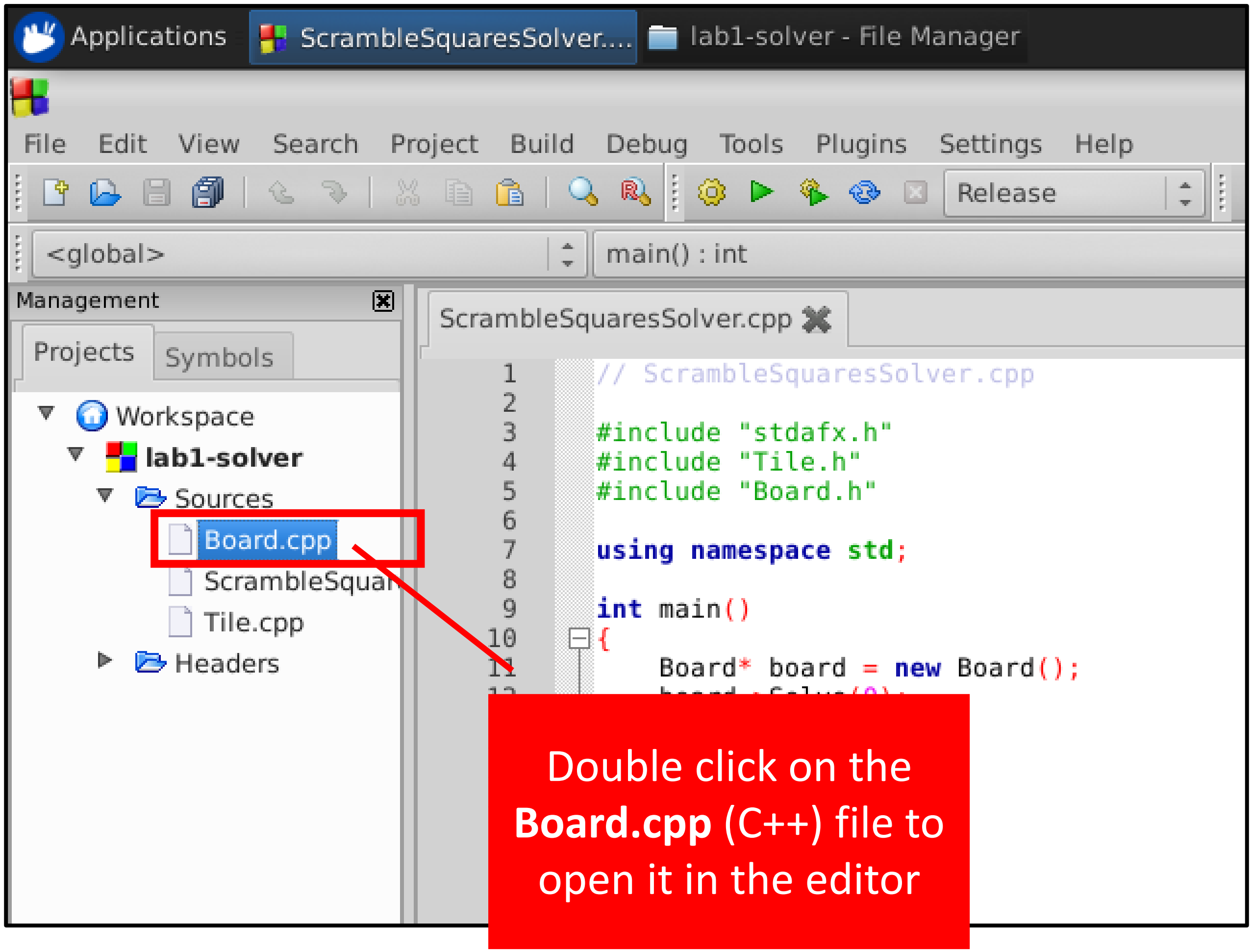
Though totally arbitrary, this initial state encoding must remain <u>consistent</u>

Encoding Bindings

- Each tile has 4 binding sites that each contain a half image value
- For each tile, you will enter your half image values in **clockwise order**
- You must provide **four** half image values for **all nine tiles**
- The sequence is North, East, South, West



Open Lab 2 – Scramble Squares Solver



Edit Lab 2 – Scramble Squares Solver

Board.cpp

```
// Board.cpp

#include "stdafx.h"
#include "Board.h"

using namespace std;

Board::Board()
{
    tiles[0] = new Tile(0, 128, 4, 2, 8);
    tiles[1] = new Tile(1, 64, 32, 2, 8);
    tiles[2] = new Tile(2, 128, 4, 32, 2);
    tiles[3] = new Tile(3, 2, 32, 4, 128);
    tiles[4] = new Tile(4, 16, 8, 64, 1);
    tiles[5] = new Tile(5, 128, 8, 16, 1);
    tiles[6] = new Tile(6, 128, 4, 16, 2);
    tiles[7] = new Tile(7, 128, 64, 2, 16);
    tiles[8] = new Tile(8, 1, 4, 16, 32);

    for(int i=0; i<9; i++)
        positions[i] = nullptr;
}
```

Update the **last four values on all 9 lines** of code to reflect your tiles

Edit file Board.cpp

```
Board::Board()
{
    tiles[0] = new Tile(0, 128, 4, 2, 8);
    tiles[1] = new Tile(1, 64, 32, 2, 8);
    tiles[2] = new Tile(2, 128, 4, 32, 2);
    tiles[3] = new Tile(3, 2, 32, 4, 128);
    tiles[4] = new Tile(4, 16, 8, 64, 1);
    tiles[5] = new Tile(5, 128, 8, 16, 1);
    tiles[6] = new Tile(6, 128, 4, 16, 2);
    tiles[7] = new Tile(7, 128, 64, 2, 16);
    tiles[8] = new Tile(8, 1, 4, 16, 32);
```

Tile Id

Creates 9 tiles but does not assign them to a position yet.

Half image value at **NORTH** binding site per initial arbitrary state

Half image value at **EAST** binding site per initial arbitrary state

Half image value at **SOUTH** binding site per initial arbitrary state

Half image value at **WEST** binding site per initial arbitrary state

- Dalmatian Head = 1
- Dalmatian Body = 2
- Shar-Pei Head = 4
- Shar-Pei Body = 8
- Spaniel Head = 16
- Spaniel Body = 32
- Shepherd Head = 64
- Shepherd Body = 128

Edit file Board.cpp

Tile Id

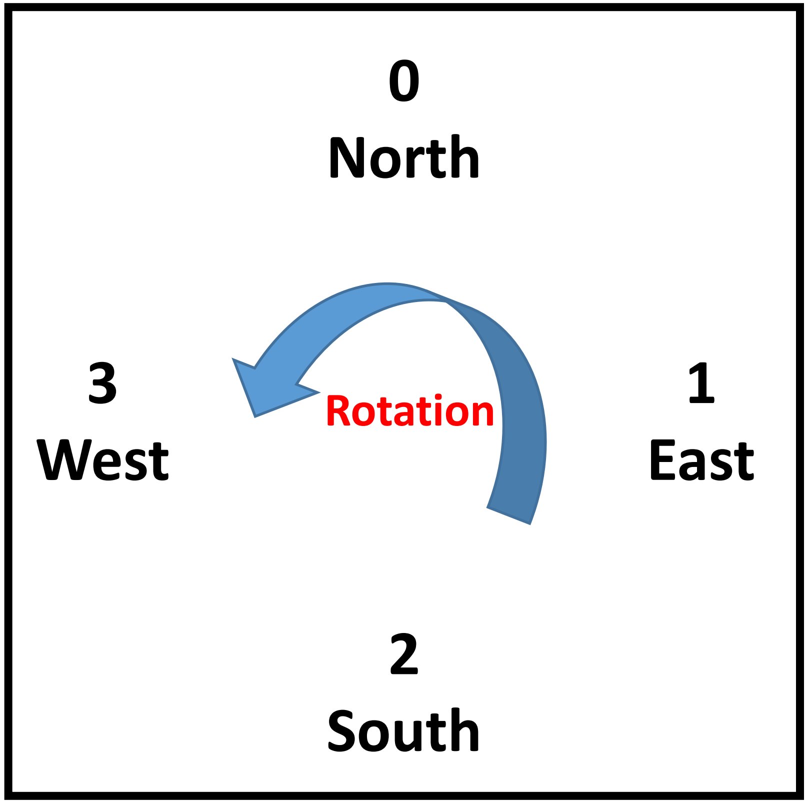
```
Board::Board()
{
    tiles[0] = new Tile(0, 128, 4, 2, 8);
    tiles[1] = new Tile(1, 64, 32, 2, 8);
    tiles[2] = new Tile(2, 128, 4, 32, 2);
    tiles[3] = new Tile(3, 2, 32, 4, 128);
    tiles[4] = new Tile(4, 16, 8, 64, 1);
    tiles[5] = new Tile(5, 128, 8, 16, 1);
    tiles[6] = new Tile(6, 128, 4, 16, 2);
    tiles[7] = new Tile(7, 128, 64, 2, 16);
    tiles[8] = new Tile(8, 1, 4, 16, 32);
```

Please **do not delete** the Tile Id # (the first number in the parentheses) for each row – **that # must remain**!

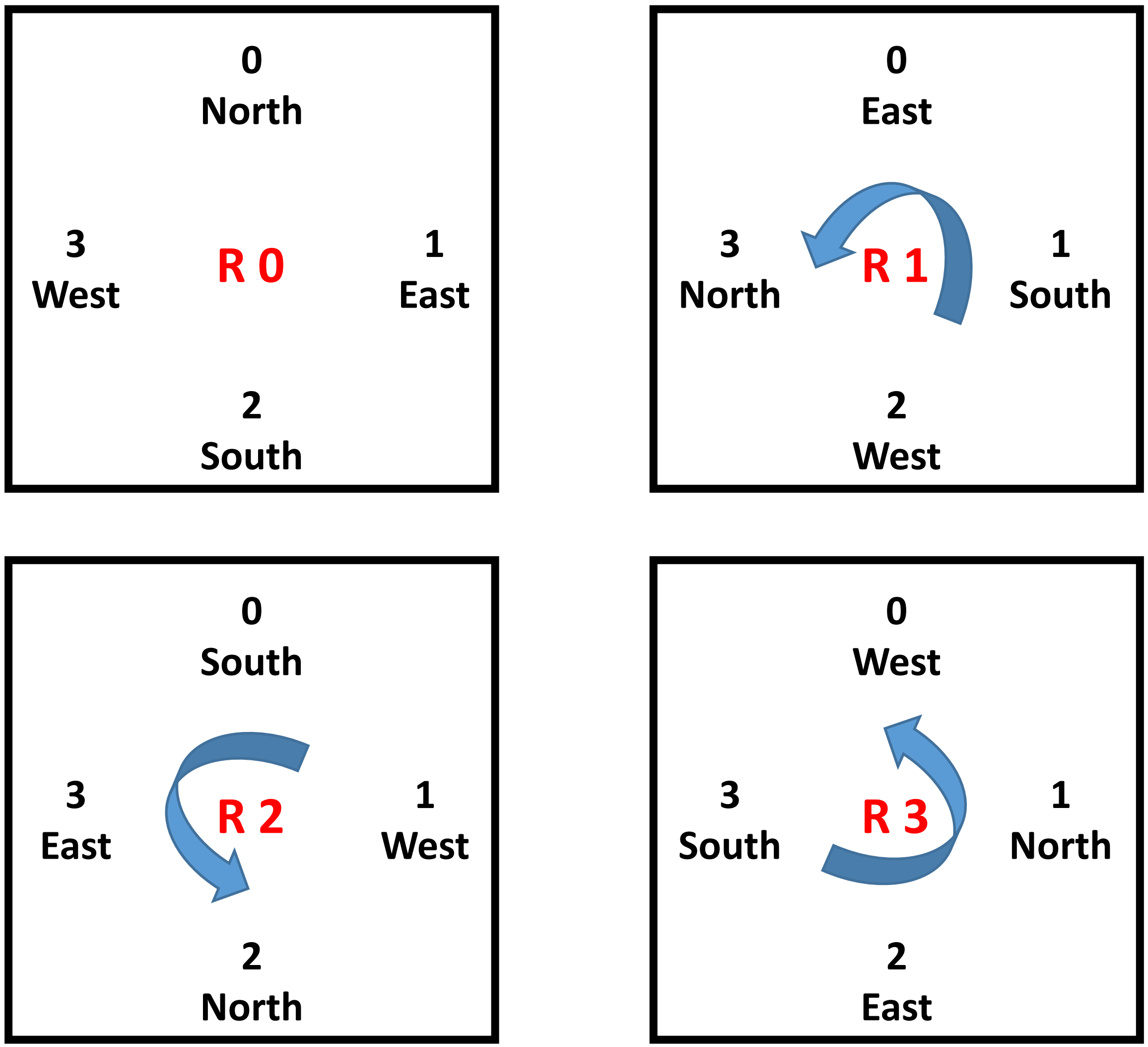
- Dalmatian Head = 1
- Dalmatian Body = 2
- Shar-Pei Head = 4
- Shar-Pei Body = 8
- Spaniel Head = 16
- Spaniel Body = 32
- Shepherd Head = 64
- Shepherd Body = 128

Encoding Bindings and Rotations

- When the program solves your puzzle, it will indicate where to place each tile # and how to **rotate** that tile
- Quarter turn **counter-clockwise** rotations are numbered **0 – 3**
- **R 0** means that this tile is *not* rotated

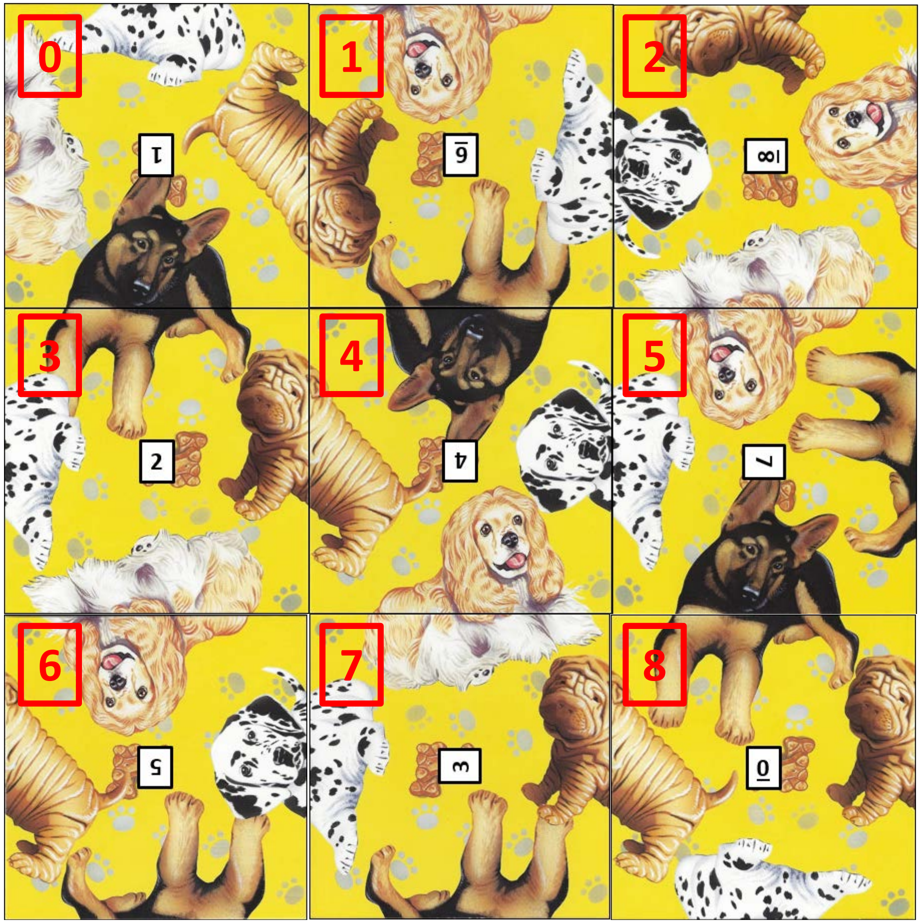


Encoding Bindings and Rotations



A Solution Written in Matrix Form

(1 r 2)	(6 r 2)	(8 r 1)
(2 r 0)	(4 r 2)	(7 r 3)
(5 r 2)	(3 r 1)	(0 r 0)



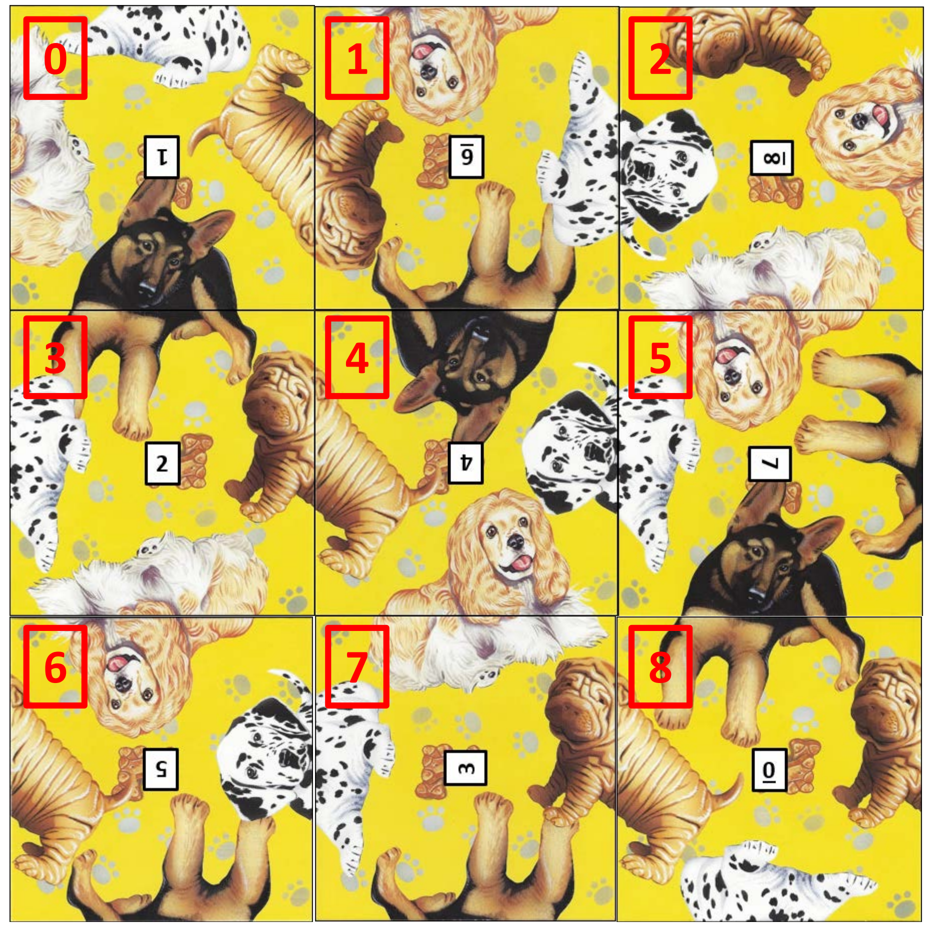
The **red squares** are the position #s – the tile #s are in the center

A Solution Written in Matrix Form

of CCW ¼ Rotations

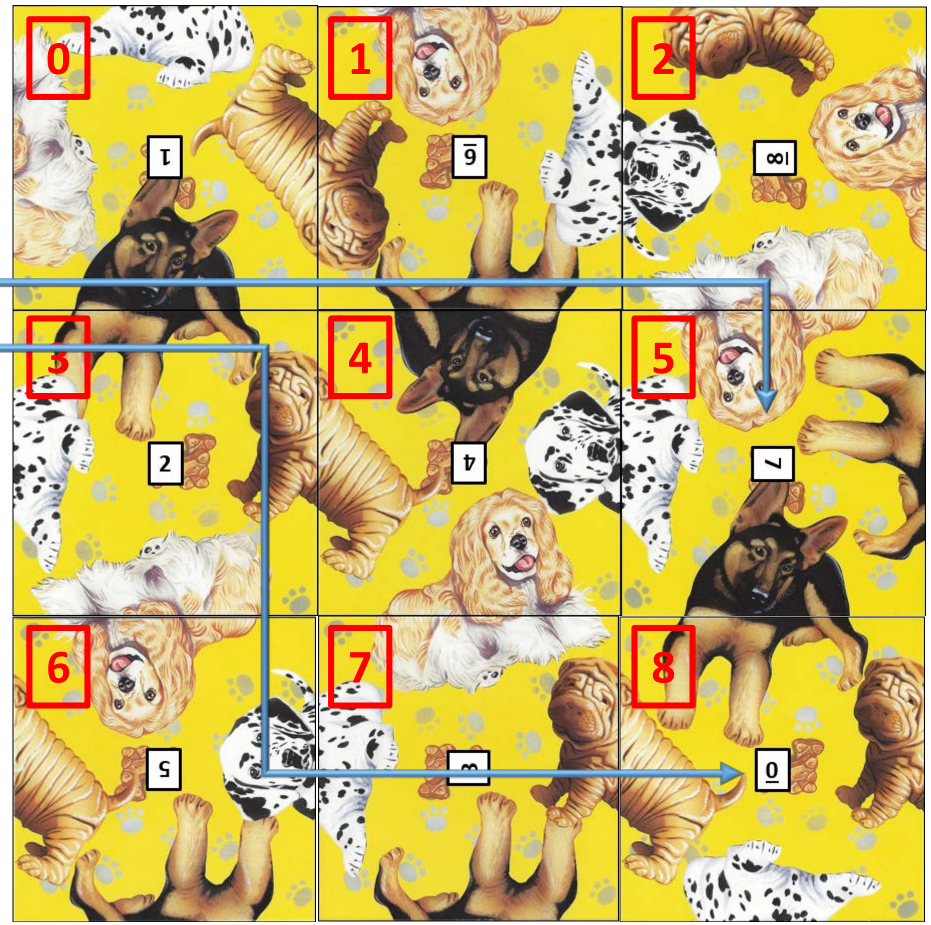
(1 r 2)	(6 r 2)	(8 r 1)
(2 r 0)	(4 r 2)	(7 r 3)
(5 r 2)	(3 r 1)	(0 r 0)

Tile Id

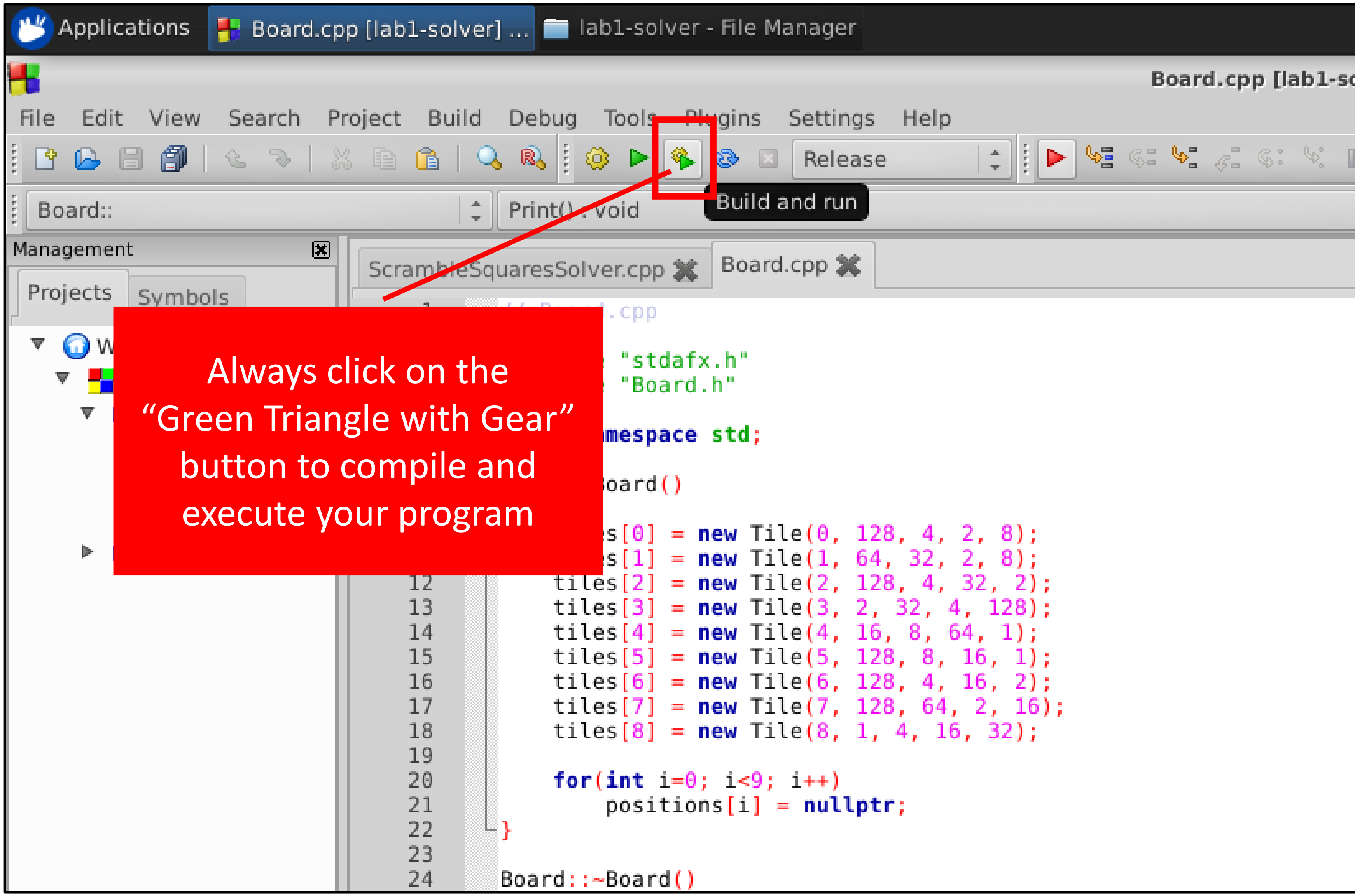


A Solution Written in Matrix Form

(1 r 2)	(6 r 2)	(8 r 1)
(2 r 0)	(4 r 2)	(7 r 3)
(5 r 2)	(3 r 1)	(0 r 0)



Run Lab 2 – Scramble Squares Solver



Check Lab 2 – Scramble Squares

lab1-solver

File Edit View Terminal Tabs Help

```
(0 r 2)   (3 r 3)   (5 r 0)
(7 r 1)   (4 r 0)   (2 r 2)
(8 r 3)   (6 r 0)   (1 r 0)

(1 r 2)   (6 r 2)   (8 r 1)
(2 r 0)   (4 r 2)   (7 r 3)
(5 r 2)   (3 r 1)   (0 r 0)

(5 r 1)   (2 r 3)   (1 r 1)
(3 r 0)   (4 r 1)   (6 r 1)
(0 r 3)   (7 r 2)   (8 r 0)

(8 r 2)   (7 r 0)   (0 r 1)
(6 r 3)   (4 r 3)   (3 r 2)
(1 r 3)   (2 r 1)   (5 r 3)


Process returned 0 (0x0)   execution time : 0.015 s
Press ENTER to continue.
```

The program found all four solutions in 15 thousandths of a second!

Using Recursion To Solve Scramble Squares

- The code recurses from position to position + 1, trying to place every tile in that position, in every rotation, that can fit the board so far.
- The code does not waste time creating & checking tile permutations which **cannot possibly work.**
- The **terminating condition** in the recursive Solve() function causes the code to "bottom out" and then "back track" up the call stack when no more tiles or rotations can be found that work in the current board layout.

A Recursive **Solve()**

```
Solve(position #)
  For each tile
    If not placed
      For each rotation
        If can place (this tile in this position #)
          Mark tile is placed
          Set position # to point to this tile
          If position # == 8
            Then Print() board
              Else Solve(position # + 1)
          Reset position # to point to NULL
          Unmark tile is placed
```

A Recursive **Solve()**

```
void Board::Solve(int position)
{
    for (int tileNum = 0; tileNum < 9; tileNum++) {
        Tile* tile = tiles[tileNum];
        if (!tile->Placed) {
            for (tile->Rotation = 0; tile->Rotation < 4; tile->Rotation++) {
                if (CanPlaceTile(tile, position)) {
                    tile->Placed = true;
                    positions[position] = tile;
                    if (position == 8)
                        Print();
                    else
                        Solve(position + 1);
                    positions[position] = nullptr;
                    tile->Placed = false;
                }
            }
        }
    }
}
```

Wouldn't **Brute Force** Be Simpler?

- When writing code to solve a problem, your first step should be to consider the brute force approach.
  - Enumerate every possible layout of tiles and rotations
  - If the current layout is valid then Print() the board
  - Keep looping until all permutations are tried
- The code might be shorter, but the run time would be **exponentially** longer.
  - Polynomial run times might be tractable if you distribute the problem across thousands of computers
  - However exponential run times cannot currently be solved by brute force. (Maybe by quantum computers…)

Now you know…

- Before diving into a problem, take the time to **form meaningful research questions** to help you analyze the situation systematically – see things as a scientist!
- Binary encoding often provides a way to have **unambiguous** input and output
- **Recursion** is a powerful concept that enables efficient search algorithms
- You cannot conquer problems whose solution space grows exponentially (**combinatorial explosion**) via **brute force** alone – you need to find a smarter way to quickly rule out permutations that cannot possibly be valid