Abstract Data Types (ADT)

Abstract Data Type

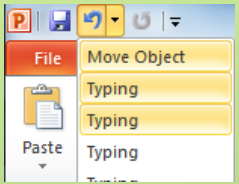
- An Abstract Data Type (ADT) is a data structure that specifies:
  - The characteristics of the collection of data
  - The operations that can be performed on the collection of data
  - But not its implementation details

Why Abstract Data Type?

- The Abstract Data Type (ADT) **hides the details of the implementation** from users
- Key advantages
  - Make programming easier
  - Do not need to re-implement the data type
  - Any changes to the underlying implementation of the ADT does not affect the usage of the data type

Stack

- Many real-life examples involve **Stack**
  - Stack of coins, stack of books, stack of food trays in cafeteria and stack of shopping baskets in supermarkets.
  - Stack of actions for the “undo” operations in a software application such as Word or PowerPoint



Stack and Queue

Stack

- Addition and removal of entries can only be carried out at **the top**
- Stack is a **Last-In-First-Out** (**LIFO**) data structure
- Two commonly operations: **push** and **pop**

Queue

- **Addition** of entries can only be carried out at **the tail**
- **Removal** of entries can only be carried out at **the head**
- Queue is a **First-In-First-Out (FIFO)** data structure
- Two commonly used operations: **addLast** and **removeFirst**

Create a Stack

- A stack can be created by using the constructor for the Stack class: Stack( )
  - Stack s = new Stack( );
- A good practice is to specify the type of objects that the stack is intended to store
  - Example:

```
Stack<Integer> intStack = new Stack<Integer>();
```

Methods in Stack

```
import java.util.Stack;
```

Method	Sample Usage

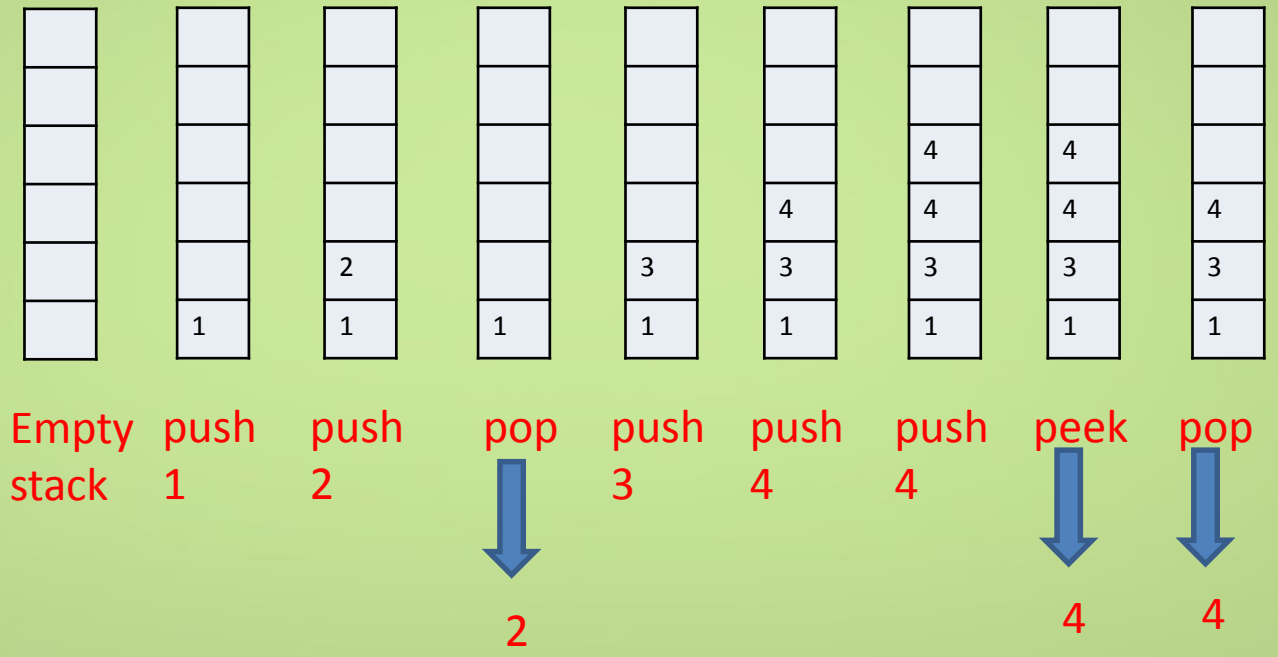
Constructor	// An empty stack of integers
	Stack<Integer> intStack = new Stack<Integer>();
	// An empty stack of floating-point numbers
	Stack<Double> doubleStack = new Stack<Double>();

push()	// Assume intStack is created already
	intStack.push(3); // push 3 to the top of the stack
	intStack.push(4); // push 4 to the top

pop()	// Assume intStack is created and it is non-empty int topValue = intStack.pop(); // remove the top element from the stack
peek()	// Assume intStack is created and it is non-empty int topValue = intStack.peek(); // look at the top element without removing it

empty()	// Check whether intStack is empty or not
	boolean isEmpty = intStack.empty();
	if (isEmpty == false) {
	  int topValue = intStack.pop();
	}

push/pop/peek examples



From decimal to binary number

- To convert a decimal number to a binary number
- An initial approach
  - For a given number n, repeat the process of
    - Find the remainder by dividing the number by 2
    - Output the remainder
    - Update n to n/2 (using integer division)
- Example: for n = 29
  - If the remainders are output in the order they were computed: 10111
  - The correct answer: 11101
  - Push the remainders onto a stack and then output the result by removing the entry on top of the stack

n = 29

n/2	remainder
14	1
7	0
3	1
1	1
0	1

Java Program

```
import java.util.Stack;
public class ToBinary {

public static Stack<Integer> s = new Stack<Integer>();

   public void outputBinary(int n) {
   while (n > 0) {
      int bit = n%2;
      s.push(Integer.valueOf(bit));
      n = n/2;
   }
   while (!s.empty()) {
        int bit = s.pop().intValue();
        System.out.print(bit);
   }
   System.out.println("");
   }
}
```

Java Program

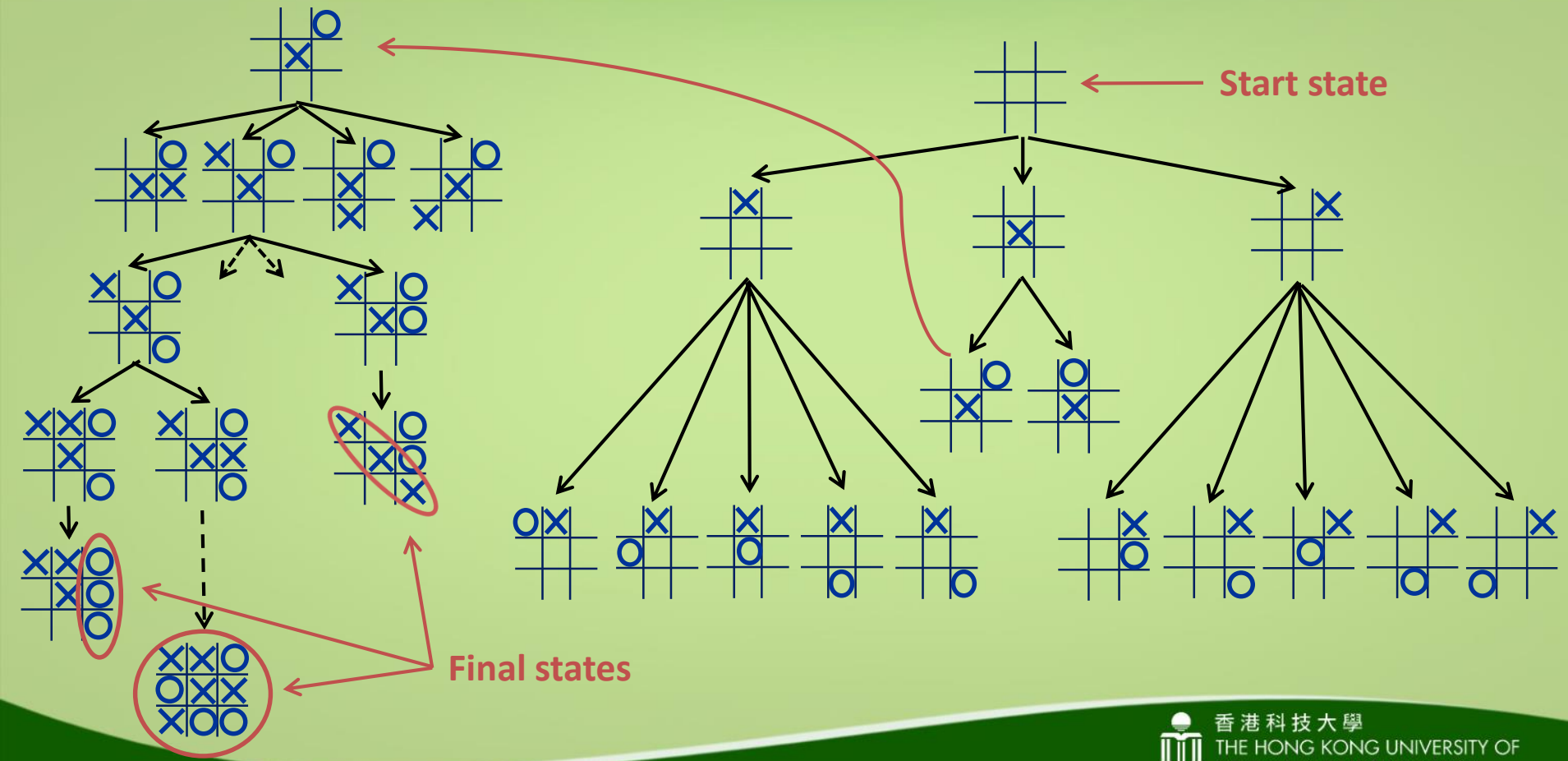
```
import java.util.Stack;
public class ToBinary {

public static Stack<Integer> s = new Stack<Integer>();

   public void outputInBinary(int n) {
   while (n > 0) {
      int bit = n%2;
      s.push(bit);  //autoboxing will convert this to s.push(Integer.valueOf(bit));
      n = n/2;
   }
   while (!s.empty()) {
        int bit = s.pop( );  //unboxing will convert this to int bit = s.pop().intValue();
        System.out.print(bit);
   }
   System.out.println("");
   }
}
```

State Space Representation

State Space Representation

- A problem is represented as a set of states
- A state space is the set of all possible states, including
  - initial states
  - final states
- Two states are connected if there is an operation that can transform one state to the other

Tic Tac Toe

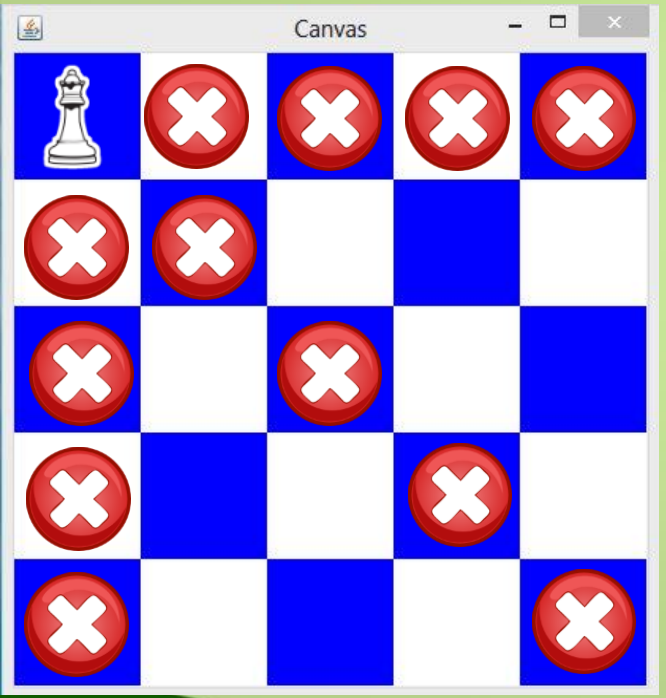


Backtracking

- **Backtracking** is a general problem strategy for searching systematically for a solution to a problem among all possible options.
- **Stacks** are often used in the implementation of backtracking algorithms.
- Example:
  - N-Queen problem
  - Aim: Place N queens on an NxN checkerboard so that no two queens can attack each other.

N-Queen Problem

N = 5

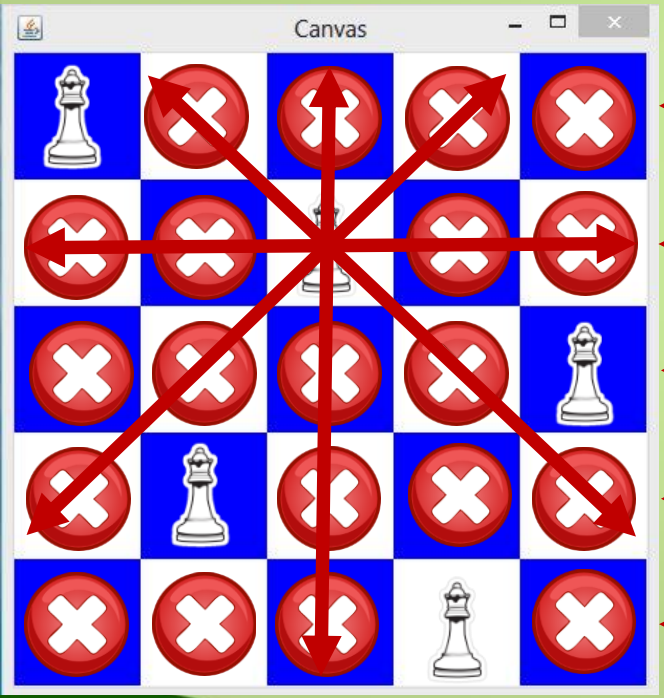


- Aim: Place N queens on an NxN checkerboard so that no two queens can attack each other.
- That is, no two queens can be on the same:
  - row
  - column
  - diagonal

N-Queen Problem

N = 5

- Aim: Place N queens on an NxN checkerboard so that no two queens can attack each other.



row = 0

row = 1

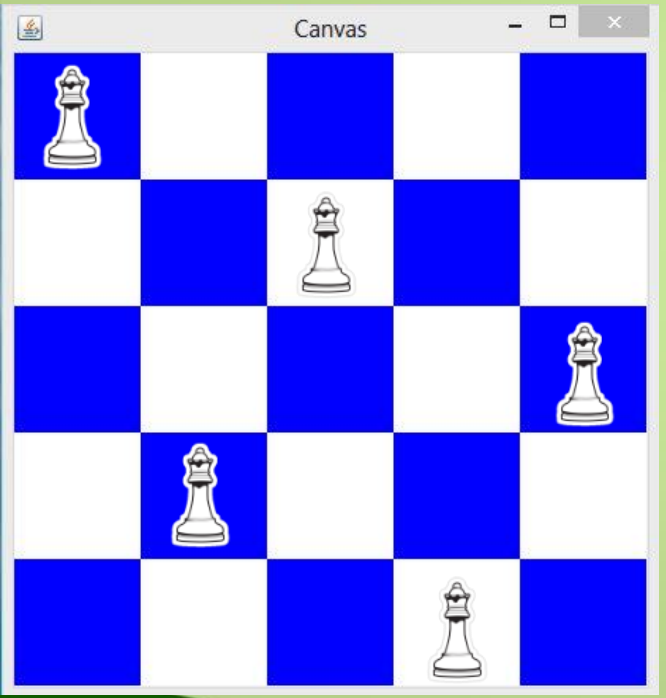
row = 2

row = 3

row = 4

N-Queen Problem

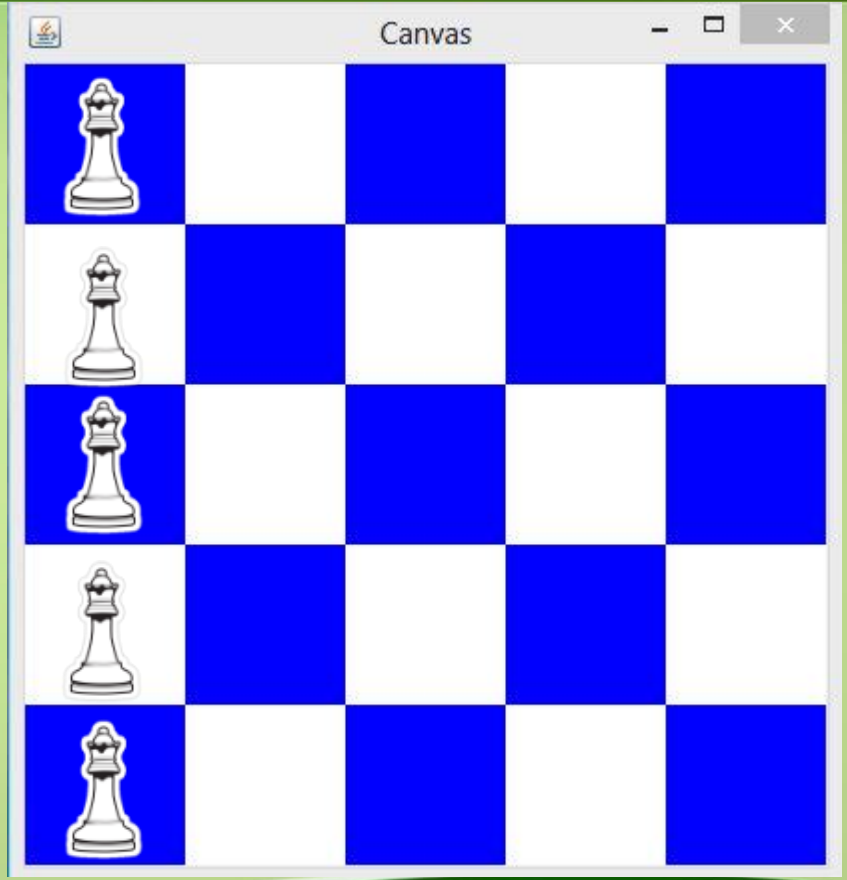
N = 5



- Aim: Place N queens on an NxN checkerboard so that no two queens can attack each other.
- That is, no two queens can be on the same:
  - row
  - column
  - diagonal

N-Queen Problem

1st Solution



3

1

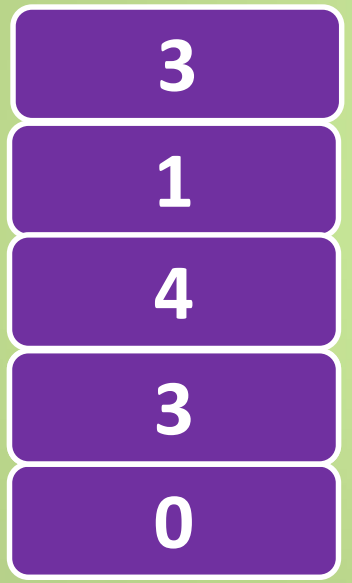
4

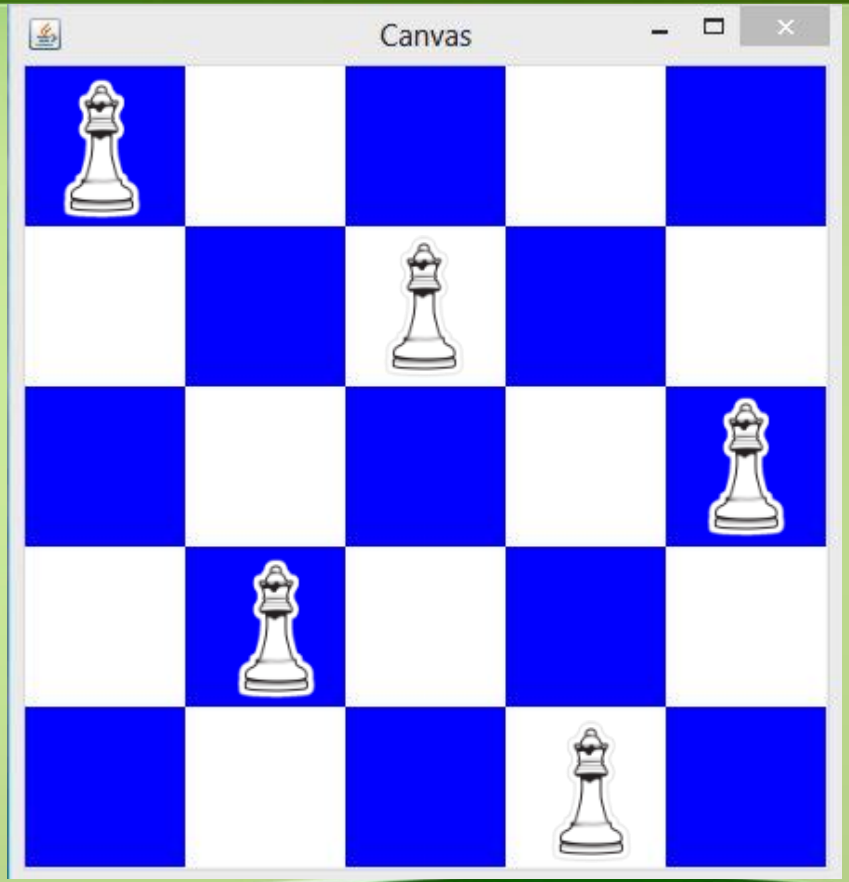
2

0

Stack

N-Queen Problem





N-Queen Problem

2nd Solution

2

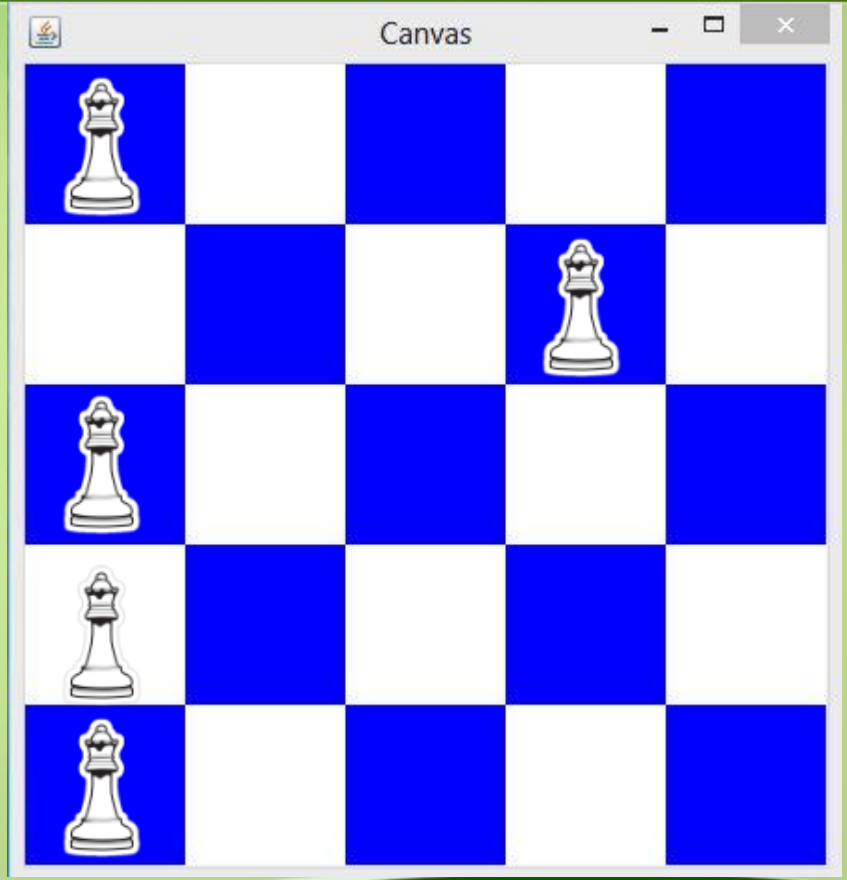
4

1

3

0

Stack



Backtracking Algorithm

- For each row on the checkerboard
  - Try placing a queen in a column that doesn't have any conflict
  - If there is no conflict, add the move to a **stack**
    else shift the queen on the next column
  - Check if a solution is found, if yes, output the solution and backtrack to find the next solution.
    // backtracking can be accomplished by popping the **stack**
  - If there is no more room to shift the queen,
    backtrack to the previous row
  - After backtracked to the previous row, shift the queen from the previous column to the next column.

Java Program

```
import java.util.Stack;

public class NQueen {
  public static Stack<Integer> s = new Stack<Integer>();
  public static int n;        // n is the number of queens
  public static int total = 0;    // total is the total number of solution.

  public static void solve(int n) {      //finds all solutions to the n-queen problem
     int row = 0;
     int col = 0;

    while ( row < n ) {      // go through each row to place a queen
      while ( col < n ) {       // go through the columns within each row
        if ( isConflict(row, col) == false ) {     // check if there is a conflict
           s.push(col);     // push col to stack
           break;           //break out of loop to next row
        }
        else
           col++;
    }
```

N-Queen Problem

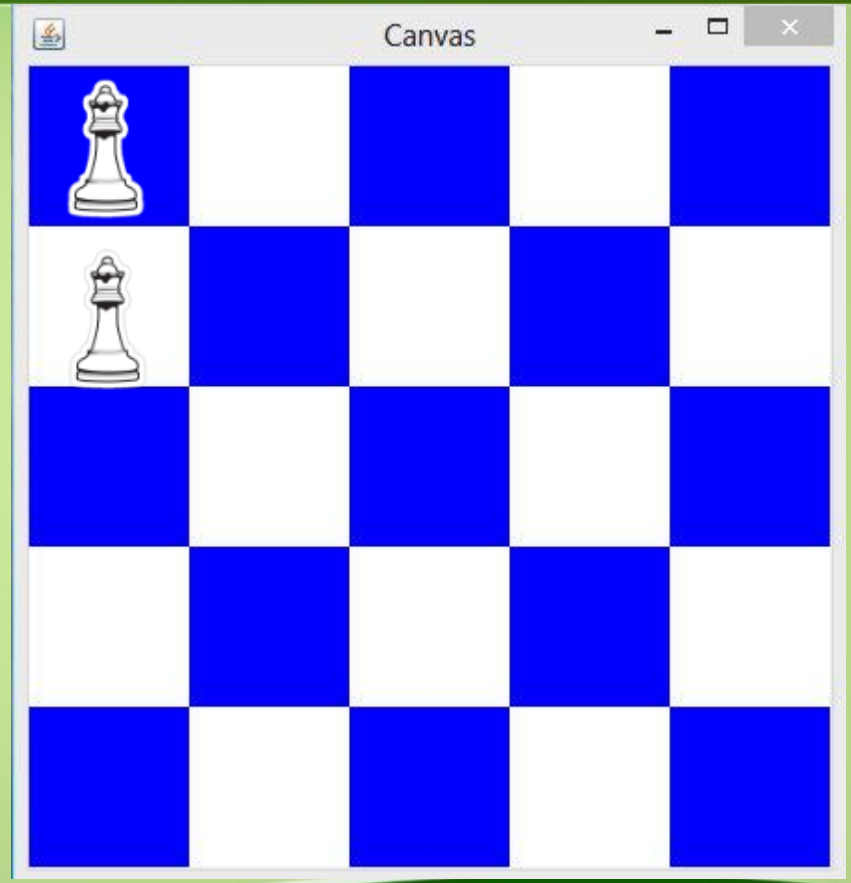
```
while ( row < n ) {
   while ( col < n ) {
      if ( isConflict(row, col) == false ) {
         s.push(col);
         break;
      }
      else   col++;
   }
... row++; col = 0;
}
```

row = 1

col = 2

true

push(2)



2

0

Stack

N-Queen Problem

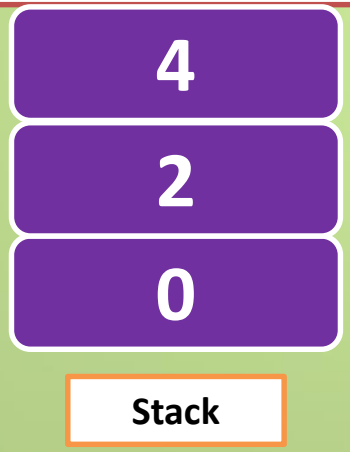
```
while ( row < n ) {
   while ( col < n ) {
      if ( isConflict(row, col) == false ) {
         s.push(col);
         break;
      }
      else   col++;
   }
  …  row++; col = 0;
}
```

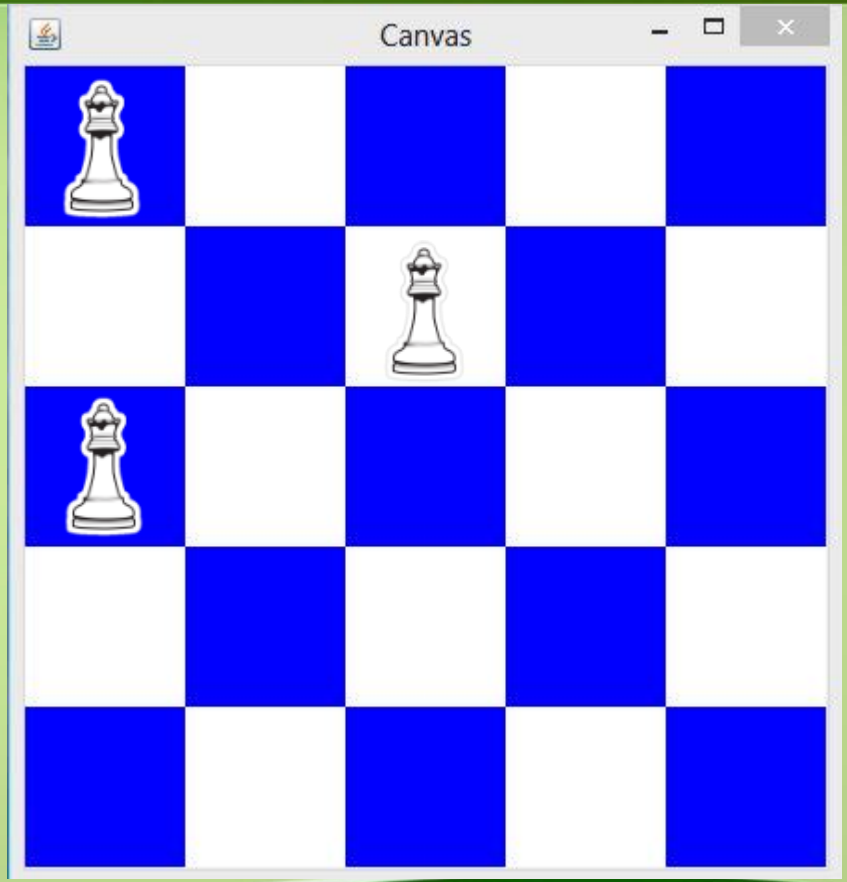
row = 2

col = 4

true

push(4)



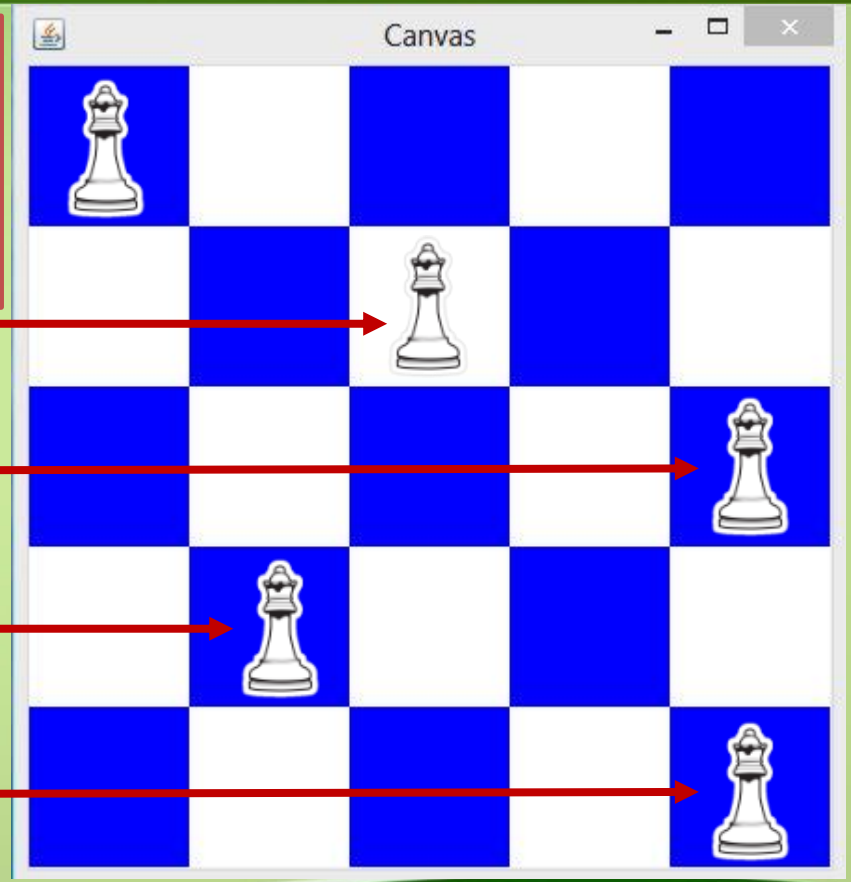


Java Program

```
    if (s.empty() == true) break; // either no solution or all solutions have been found

    if (col >= n) {            // finished all possible placements in a row
       row--;
       col = s.pop() + 1;
    }
    else {
      row++;
      col = 0;
    }
    if (s.size()==n){          // if stack size is n a solution is found
         total++;
         System.out.println(total + ": " + s);
         col = s.pop() + 1; // continue to find next solution
         row--;
       }
   }
}
```

N-Queen Problem

```
if (col >= n) {   // finished all possible placements in a row
    row--;
    col = s.pop() + 1;
}
 else {   row++;  col = 0;
}
```



1

4

3

0

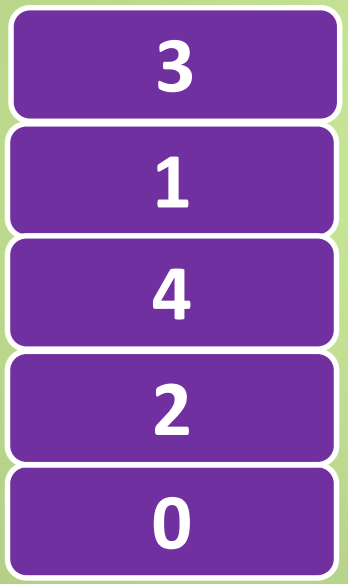
Stack

Java Program

```
        if (s.empty() == true) break;  // either no solution or all solutions have been found

        if (col >= n) {              // finished all possible placements in a row
           col = s.pop() + 1;
           row--;
         }
        else {
          row++;
          col = 0;
        }
        if (s.size()==n){          // if stack size is n a solution is found
            total++;
            System.out.println(total + ": " + s);
            col = s.pop() + 1; // continue to find next solution
            row--;
          }
    }
}
```

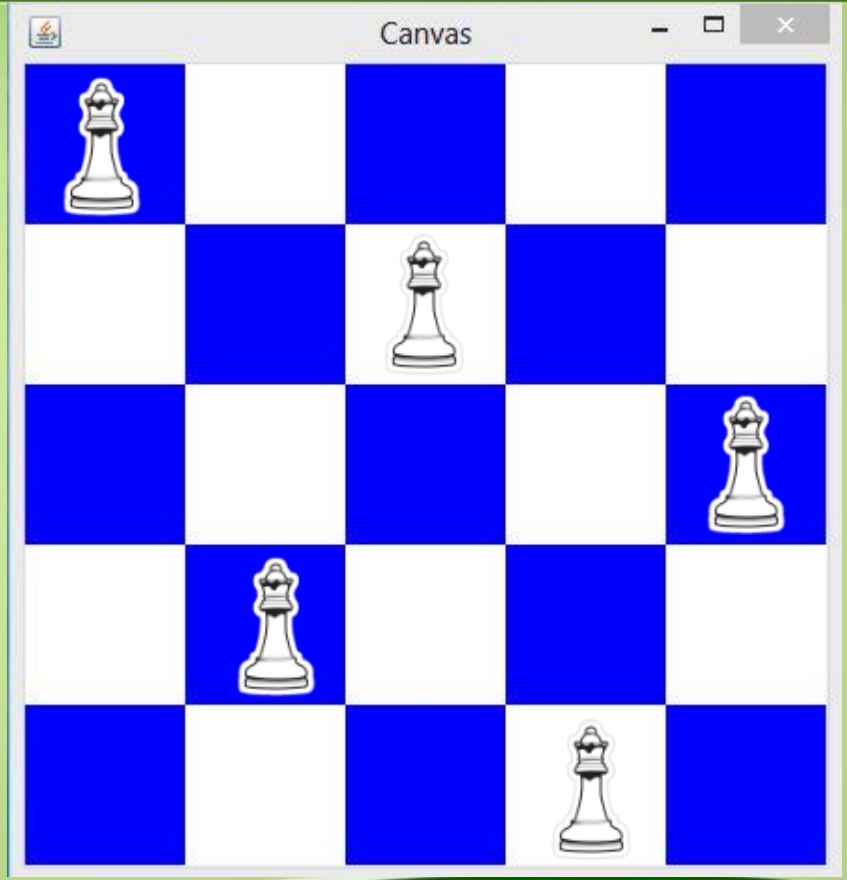
N-Queen Problem

```
if (s.size()==n){       // if stack size is n a solution is found
        total++;
        System.out.println(total + ": " + s);
        col = s.pop() + 1; // continue to find next solution
        row--;
}
```



Stack

1: [0, 2, 4, 1, 3]

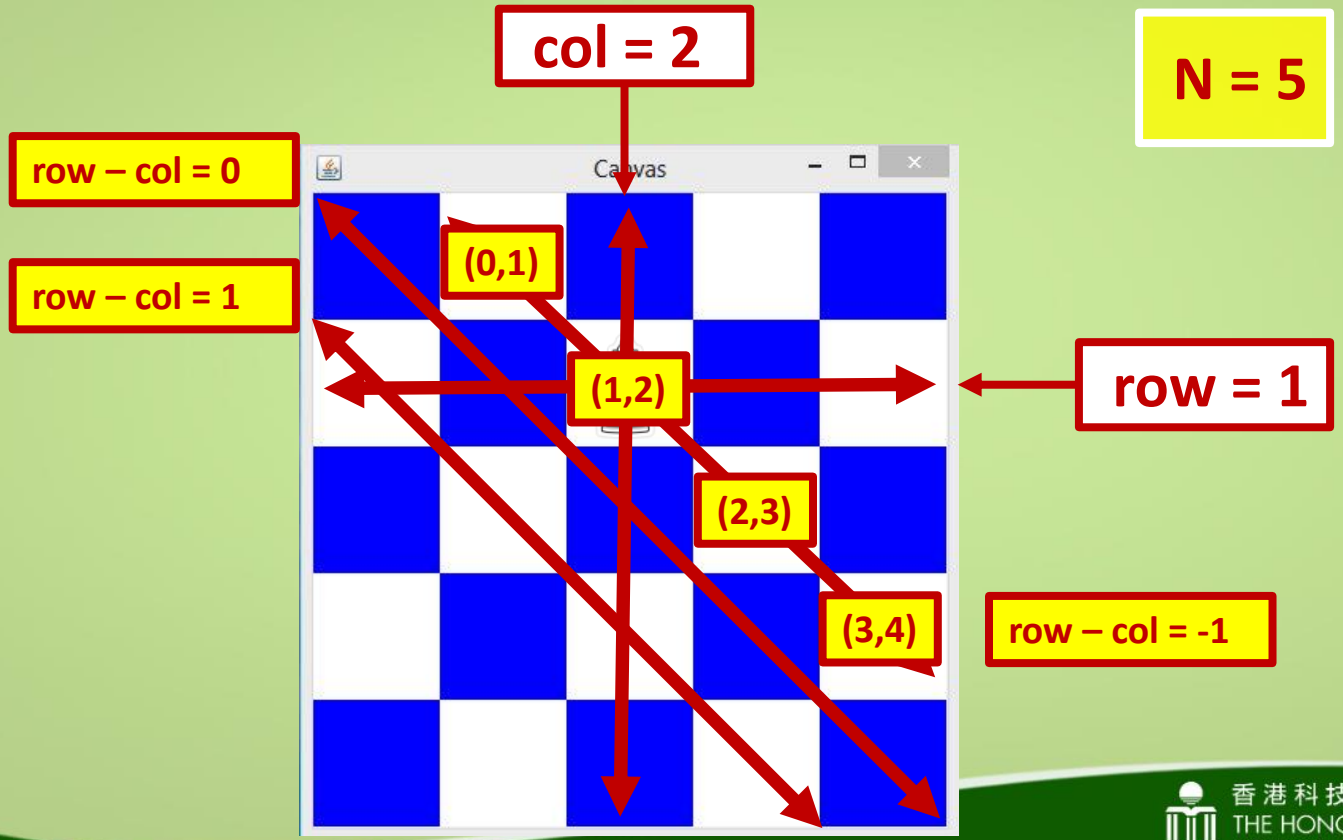


Java Program

```
public static boolean isConflict(int row, int col) {
    int diff = row-col;
    int sum  = row+col;
    for (int i = 0; i < row; i++) {
        int t = s.get(i);
        if (t==col || i-t == diff || i+t == sum) return true;
    }
    return false;
}
```

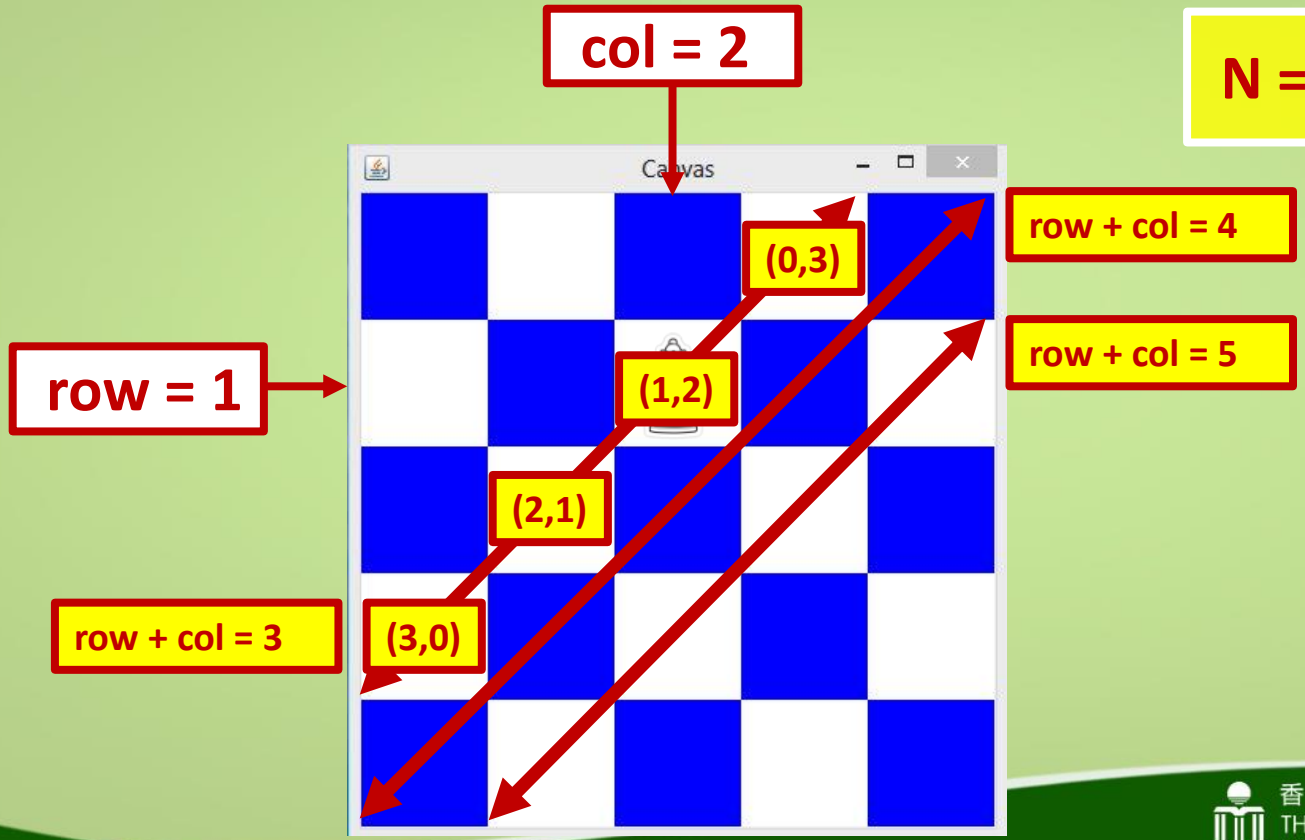
N-Queen Problem

- How to determine if there is a conflict?



N-Queen Problem

- How to determine if there is a conflict?



Java Program

```
public static boolean isConflict(int row, int col) {
    int diff = row-col;
    int sum  = row+col;
    for (int i = 0; i < row; i++) {
        int t = s.get(i);
        if (t==col || i-t == diff || i+t == sum) return true;
    }
    return false;
}
```

N-Queen Problem

3rd Solution

4

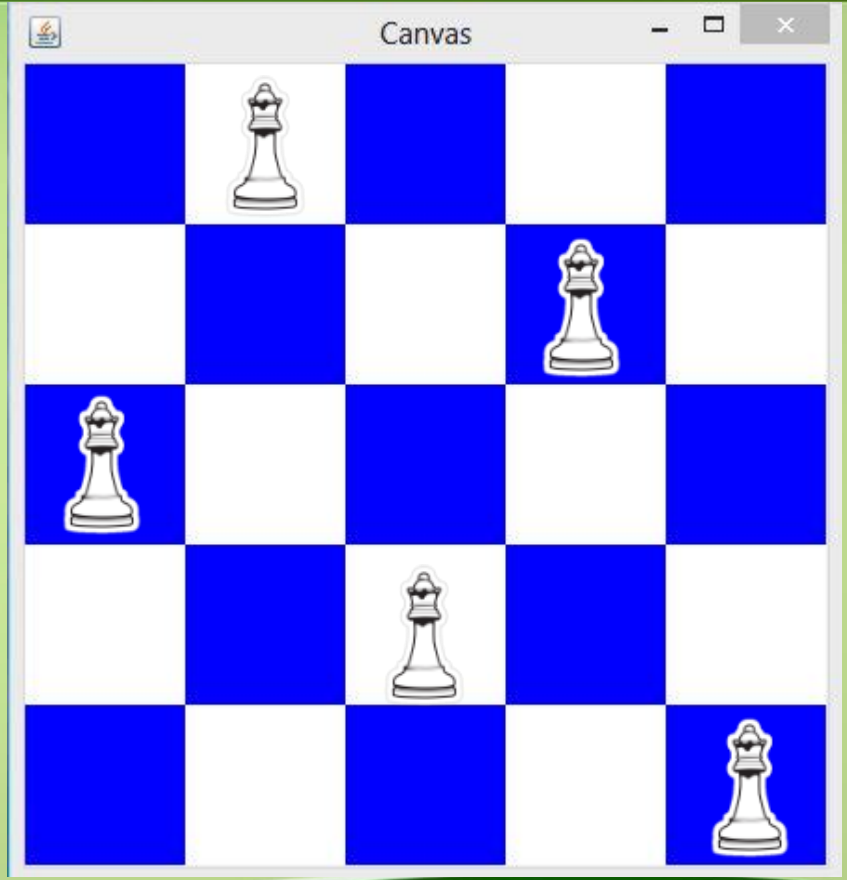
2

0

3

1

Stack



N-Queen Problem

4th Solution

3

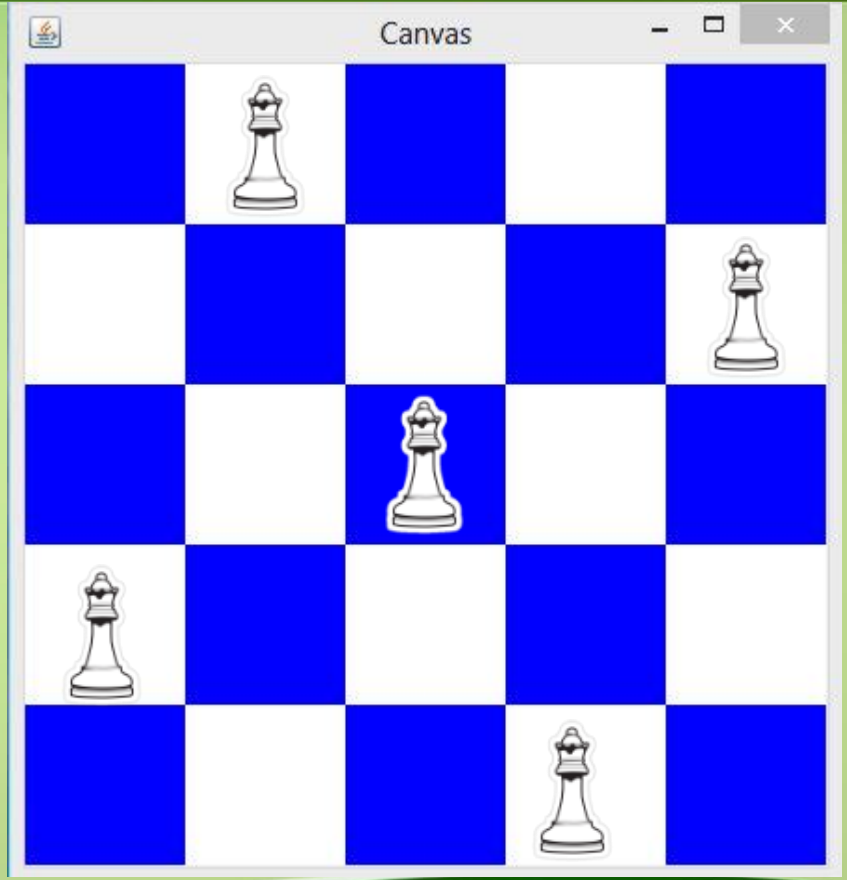
0

2

4

1

Stack

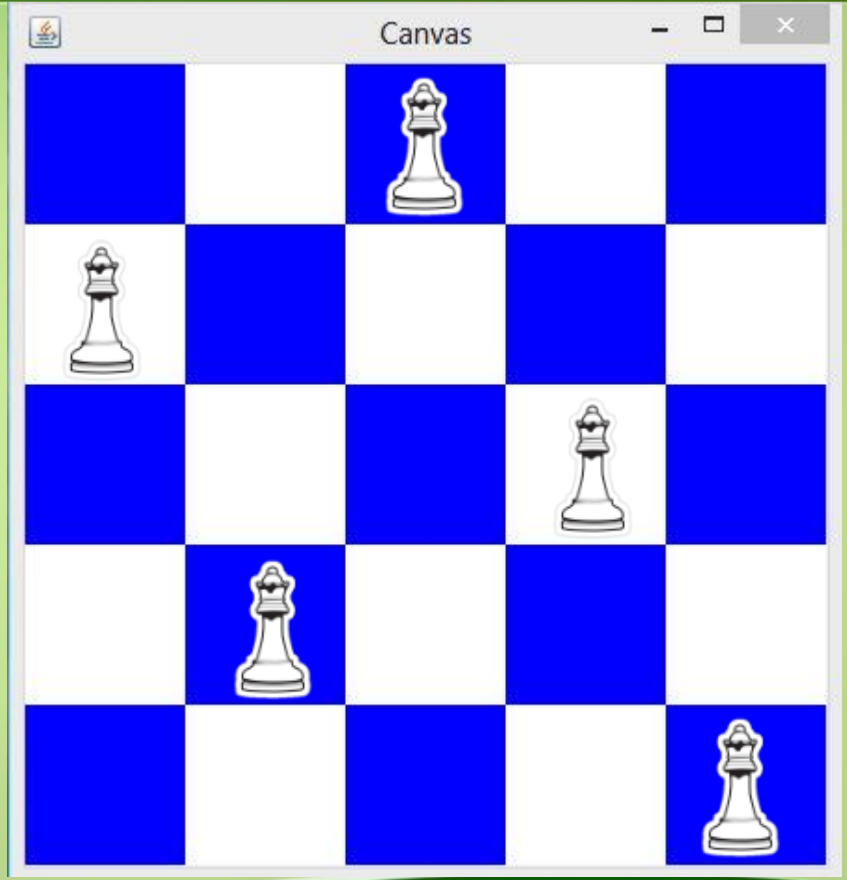


N-Queen Problem

5th Solution

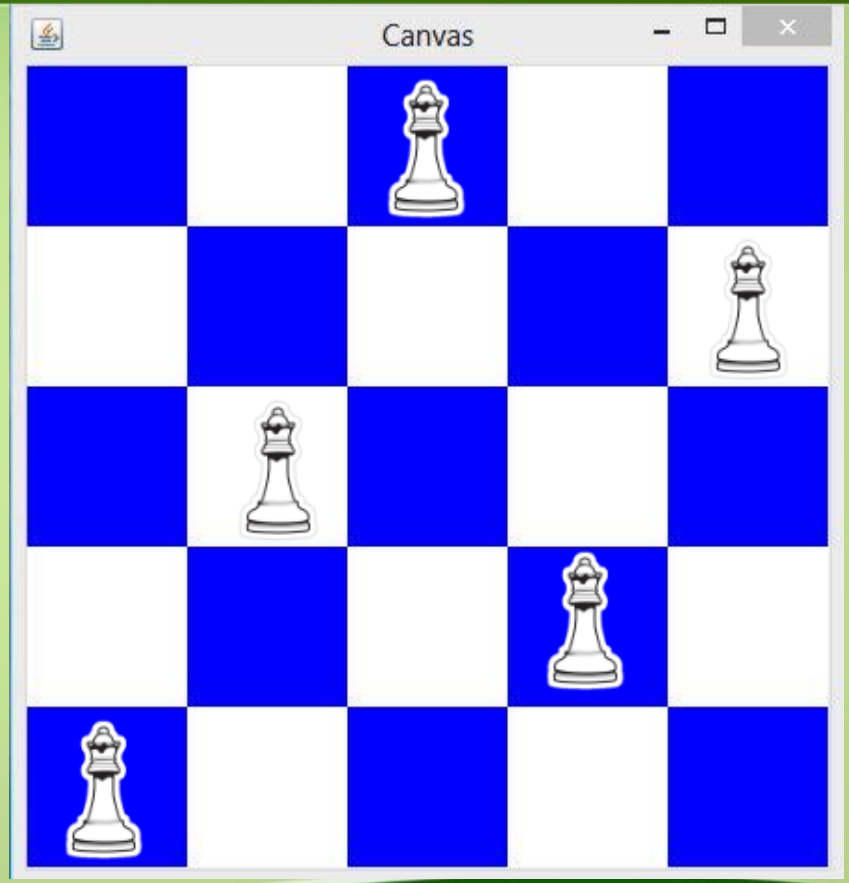
4
1
3
0
2

Stack



N-Queen Problem

6th Solution



0

3

1

4

2

Stack

N-Queen Problem

7th Solution

1

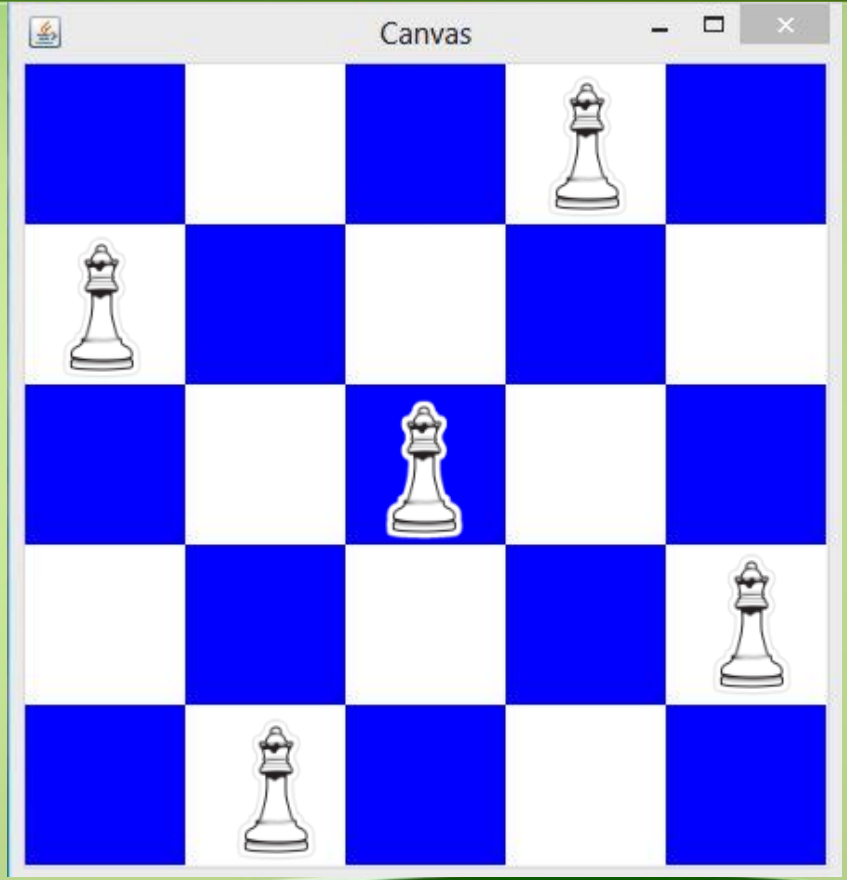
4

2

0

3

Stack

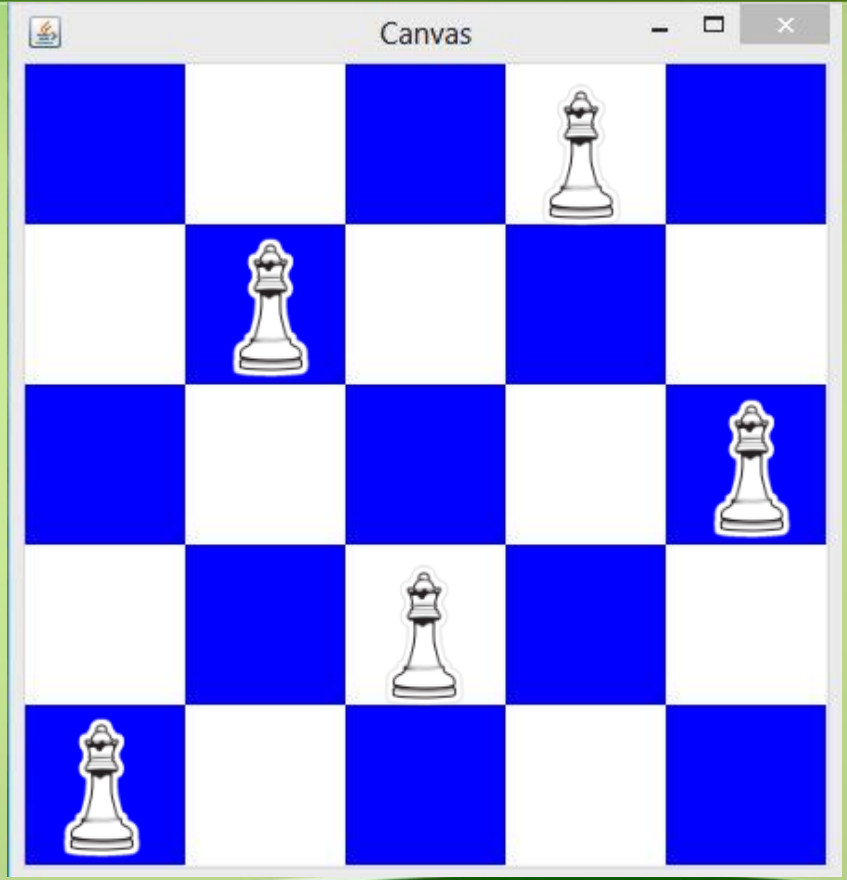


N-Queen Problem

8th Solution

0
2
4
1
3

Stack



N-Queen Problem

9th Solution

2

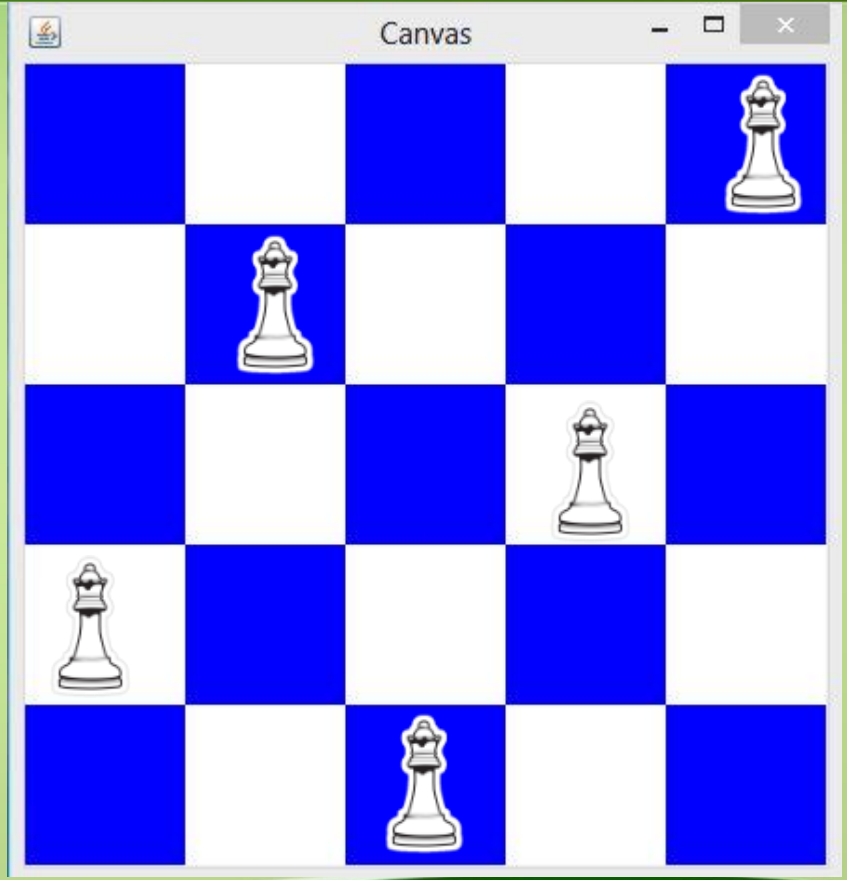
0

3

1

4

Stack



N-Queen Problem

10th Solution

1

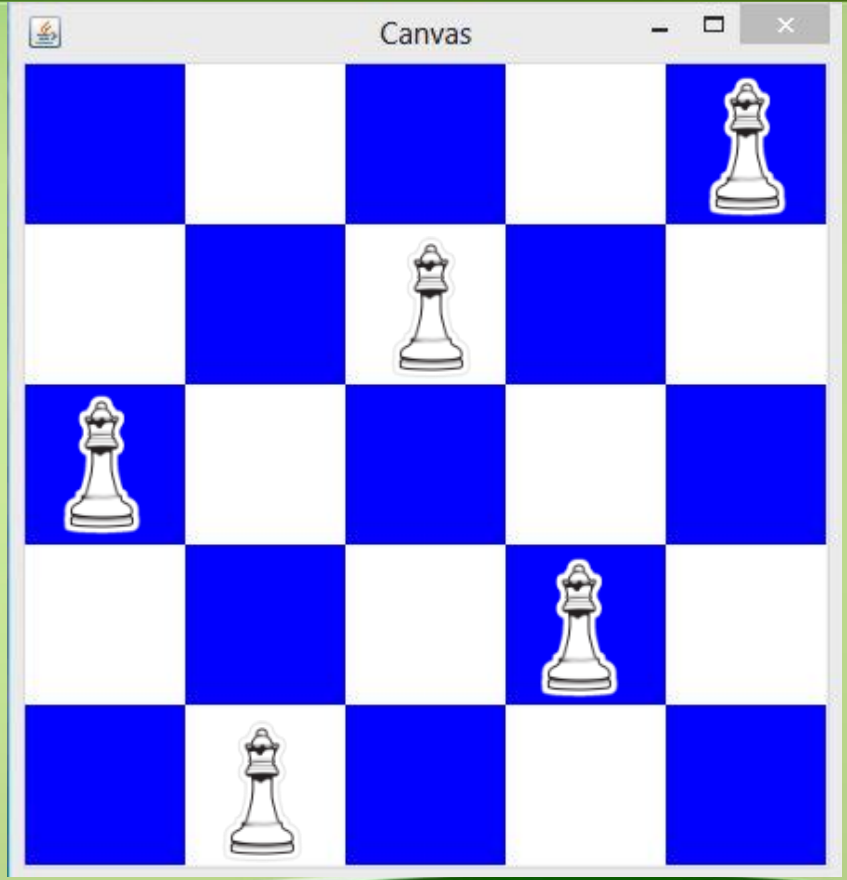
3

0

2

4

Stack

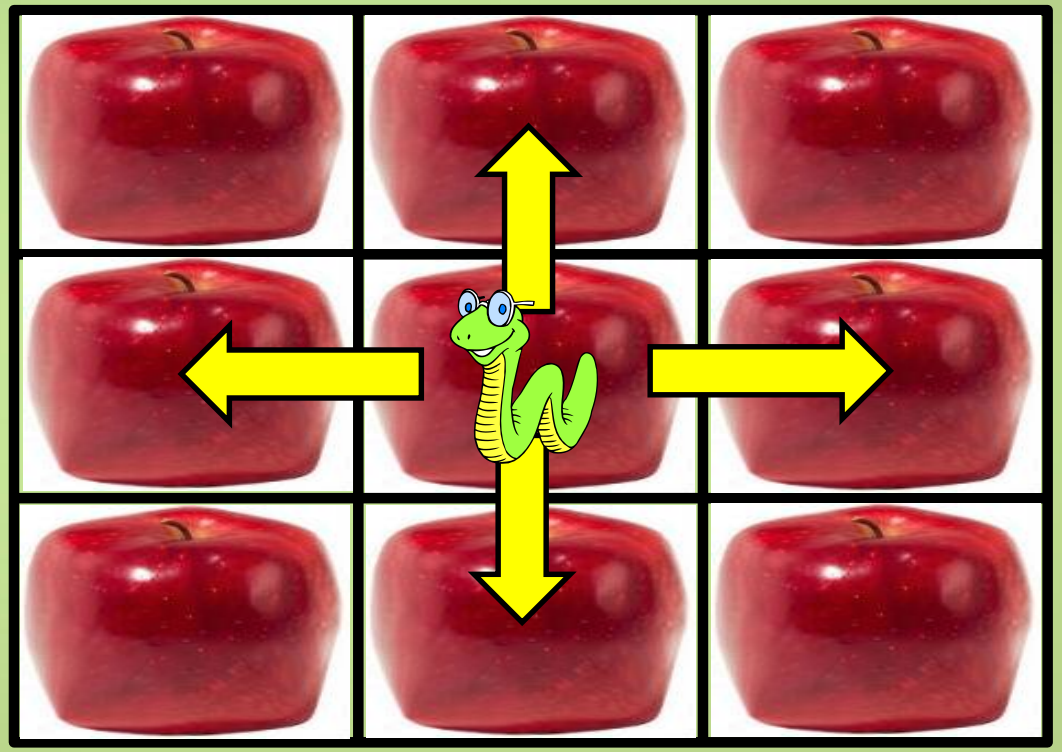


Square Apple Problem

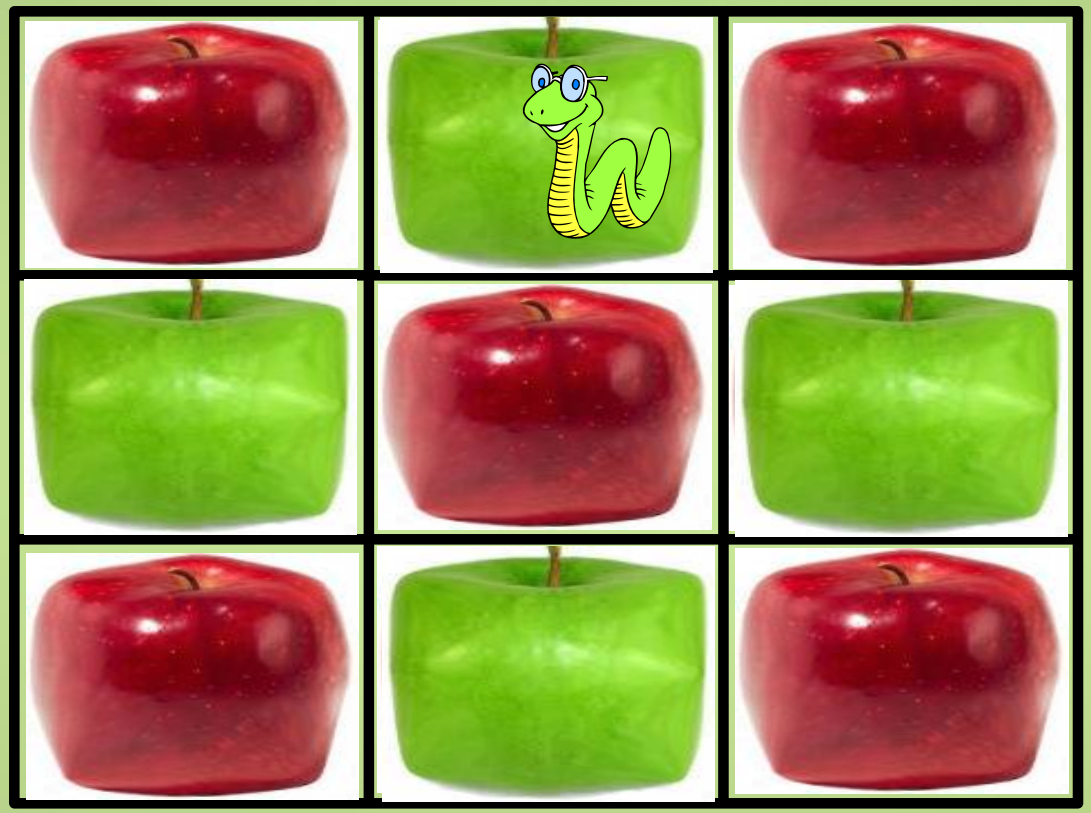
Starting from the middle cell, would it be possible for the worm to finish eating all the apples?

Rules:

- **The worm can only move into another cell that shares a common wall; and**
- **a cell that has not been previously visited.**



2D Square Apple Problem



Queue

- In daily life, we line up for various reasons.
- Unlike Stack, Queue is a **First-In-First-Out** (**FIFO**) data structure
  - **Addition** of entries can only be carried out at **the tail**
  - **Removal** of entries can only be carried out at **the head**

Useful methods in ArrayDeque

import java.util.ArrayDeque;

Method	Sample Usage

Constructor	// An empty stack of integers
	ArrayDeque<Integer> intQueue = new ArrayDeque<Integer>();
	// An empty stack of floating-point numbers
	ArrayDeque<Double> doubleQueue = new ArrayDeque<Double>();

addLast(e)
or offerLast(e)
	// Assume intQueue is created
	intQueue.addLast (50); // add 50 to the end of the queue
	intQueue.offerLast(10); // add 10 to the end of the queue and return true
	// after the insertion

removeFirst()
or pollFirst()
	// Assume intQueue is created and it is non-empty
	int firstValue = intQueue.removeFirst(); // remove the first element
	Int firstValue = intQueue.pollFirst(); // remove the first element

isEmpty()	// Boolean method to check whether intQueue is empty
	if (!intQueue.isEmpty()) {
	int firstValue = intQueue.removeFirst();
	}

addLast/removeFirst examples

```
import java.util.ArrayDeque;
public class QueueDemo {
  private ArrayDeque<Double> queue  =
                        new ArrayDeque<Double>();

  public void QueueDemo() {
          queue.addLast(4.0);
          queue.addLast(3.0);
          queue.addLast(5.0);
          if (!queue.isEmpty()) queue.removeFirst();
          queue.addLast(5.0);
          if (!queue.isEmpty()) queue.removeFirst();
          if (!queue.isEmpty()) queue.removeFirst();
          if (!queue.isEmpty()) queue.removeFirst();
   }
}
```