RECURSION

Lecture 6: Recursion

You are now Java experts!

- This was almost all the Java that we will teach you in this course
- Will see a few last things in the remainder of class
- Now will begin focusing on data-structures

Question 10 on Homework

How can you access a private field from a subclass?

-> I got it wrong. You can only use **super** if they are **inner classes** as well as extending each other

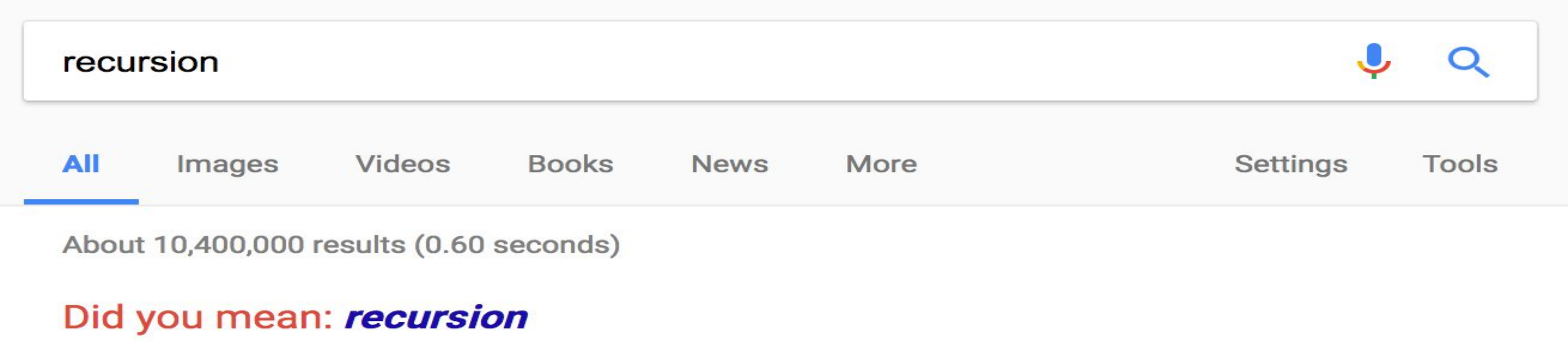
-> You can use **reflection**

Ignore that question. I'll post a note on Piazza with the details.

This Lecture

- Recursion in the wild
- How is recursion implemented
- Recursion Structure
- How to design a recursive method
- Examples of recursive problems
- Backtracking Recursion
- Examples of backtracking recursive problems

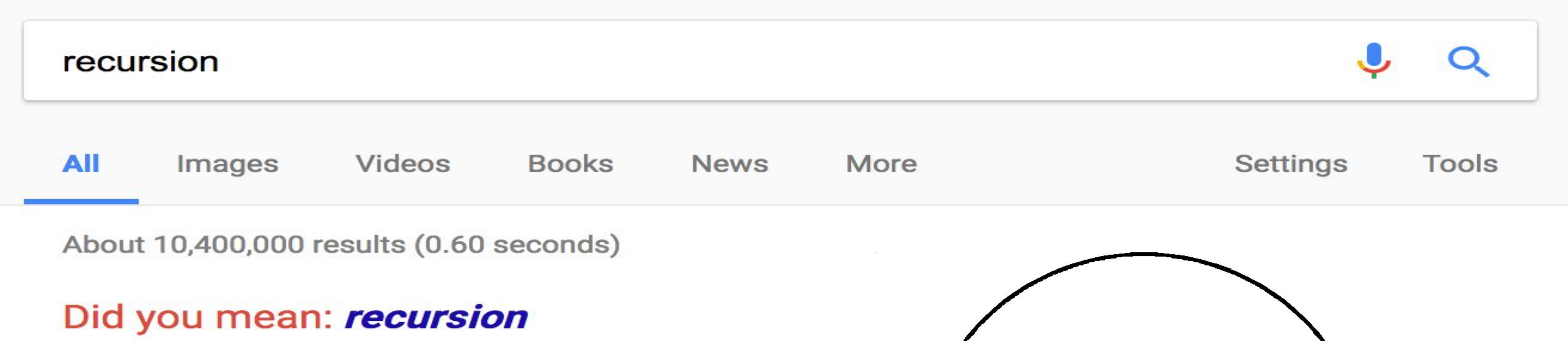
To Understand Recursion…



About 10,400,000 results (0.60 seconds)

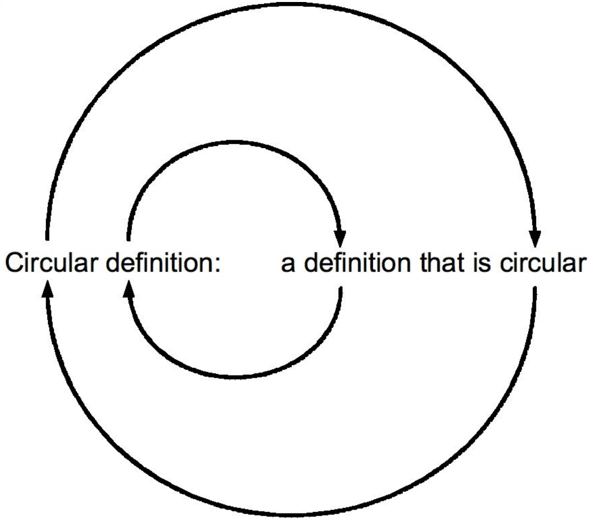
Did you mean: ***recursion***

To Understand Recursion…



About 10,400,000 results (0.60 seconds)

Did you mean: *recursion*



Definition: defining a property/functionality in terms of itself

To Understand Recursion…

- □ Break up problem into one or more **smaller subproblems** of similar structure.
- □ Solve subproblems using same method.
- □ Reach a stage where you know the answer to the sub-problem
- □ Combine results to produce solution to original problem.

Why recursion?

- Useful programming paradigm
  - Especially useful to manipulate trees, lists, collections
- Introduces the **divide-and-conquer** principle
  - When one problem is too hard, break it down into smaller subproblems, and keep doing that until you know how to solve the subproblem

Roadmap

- We’ll first look at examples of recursion in real world, in maths, in Java
- We’ll then derive from them how to write recursive methods
- We’ll look at some more examples

Recursion is real!

- My ancestors are:
  - My **parents**, and the ancestors of my parents.

Parents

Recursion is real!

- My ancestors are:
  - My **parents**, and the ancestors of my parents.
  - What are the ancestors of my parents?
    - Their **parents**, and the ancestors of their parents

Parents

+

Parents

Recursion is real!

- My ancestors are:
  - My **parents**, and the ancestors of my parents.
  - What are the ancestors of my parents?
    - Their **parents**, and the ancestors of their parents
      - What are the ancestors of their parents?
        - Their **parents**, and the ancestors of their parents …

Parents

+

Parents

+

Parents

Recursion is real!

- Factorials are defined recursively:
  - 0! = 1 n! = n x (n-1)!

- Power of a number is defined recursive
  - b^0 = 1
  - b^c = b * b ^(c-1)

Recursion is real!

- Factorials are defined recursively:
  - 0! = 1 n! = n x (n-1)!

5! = 5 x 4!

4! = 4 x 3!

3! = 3 x 2!

2! = 2x 1!

1! = 1 x 0!

1

5 x 4 x 3 x 2 x 1 = 120

- Power of a number is defined recursive
  - b^0 = 1
  - b^c = b * b ^(c-1)

Recursion is real!

- Factorials are defined recursively:
  - 0! = 1 n! = n x (n-1)!

5! = 5 x 4!

4! = 4 x 3!

3! = 3 x 2!

2! = 2x 1!

1! = 1 x 0!

1

5 x 4 x 3 x 2 x 1 = 120

- Power of a number is defined recursive
  - b^0 = 1
  - b^c = b * b ^(c-1)

2^3 = 2 * 2^2

2^3 = 2 * 2^2

2^2 = 2 * 2^1

2^1 = 2 * 2^0

1

2 * 2 * 2 * 1 = 8

Two ways to understand recursion

1. How is it executed? (or, why does this even work?)

2. How do we understand recursive methods? (or, how do we write/develop recursive methods?)

Recursion in Java

How to compute the sum of all the digits in an integer

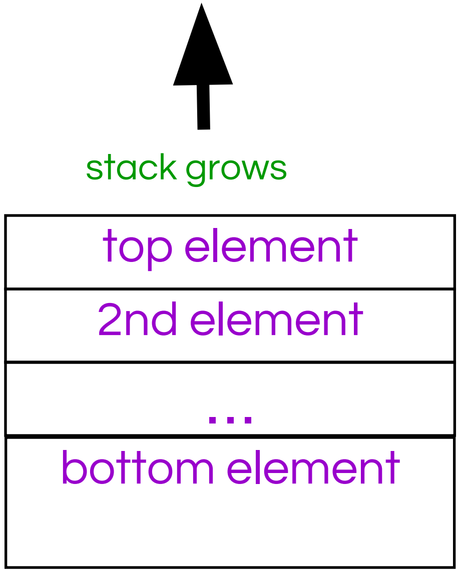
Ex: sum(1) = 1 sum(13) = 4 sum(852) = 15

```
/** = sum of digits in n.
  * Precondition:  n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;

   // { n has at least two digits }
   // return first digit + sum of rest
   return n%10  +  sum(n/10);
}
```

sum calls itself!

An implementation detour: Stacks and Queues



Stack: list with (at least) two basic ops:

* Push an element onto its top

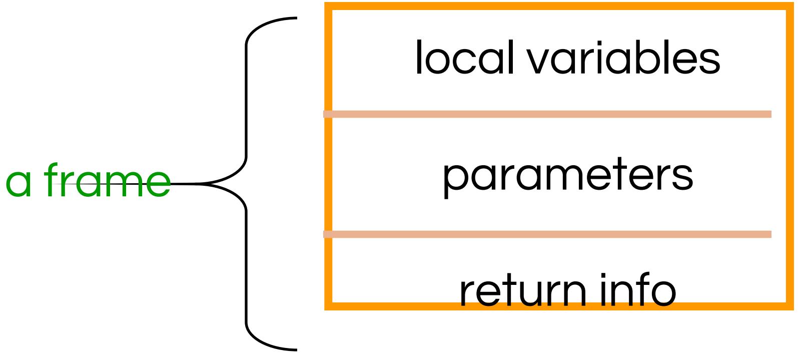
* Pop (remove) top element

Last-In-First-Out (LIFO)

Like a stack of trays in a cafeteria

An Implementation Detour: Stack Frame

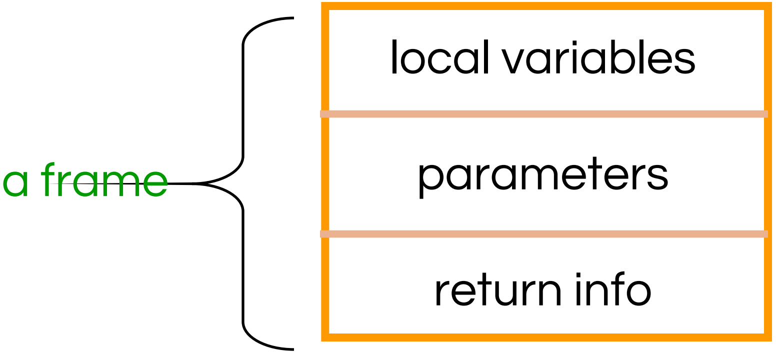
A **frame** contains information about a method call:



An Implementation Detour: Stack Frame

A **frame** contains information about a method call:

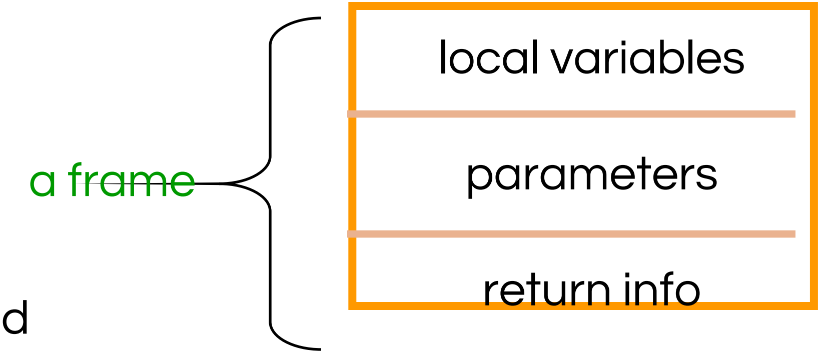
At runtime Java maintains a **stack** that contains frames for all method calls that are being executed but have not completed.



An Implementation Detour: Stack Frame

A **frame** contains information about a method call:

At runtime Java maintains a **stack** that contains frames for all method calls that are being executed but have not completed.

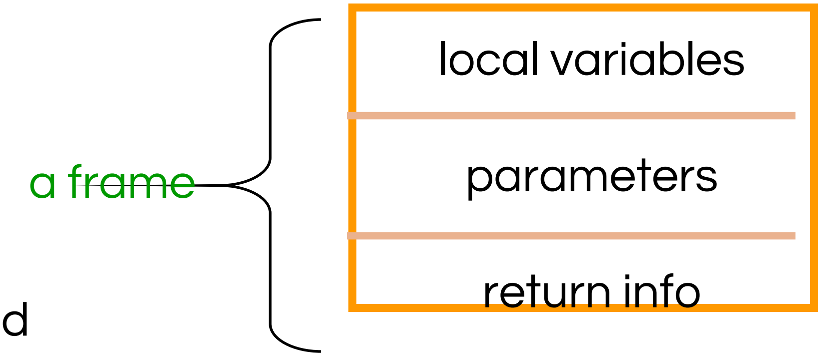


Start of method call: push a frame for call on stack. Use the frame for the call to reference local variables and parameters.

An Implementation Detour: Stack Frame

A **frame** contains information about a method call:

At runtime Java maintains a **stack** that contains frames for all method calls that are being executed but have not completed.



Start of method call: push a frame for call on stack. Use the frame for the call to reference local variables and parameters.

End of method call: pop its frame from the stack; if it is a function leave the return value on top of stack.

An implementation detour: Memorise method call execution!

A frame for a call contains parameters, local variables, and other information needed to properly execute a method call.

To execute a method call:

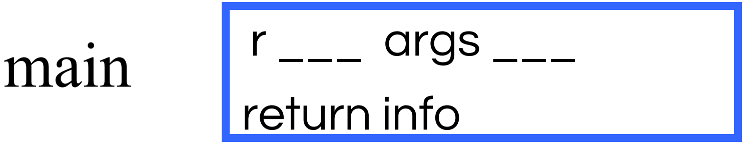
1. push a frame for the call on the stack,
2. assign argument values to parameters,
3. execute method body,
4. pop frame for call from stack, and (for a function) push returned value on stack

When executing method body look in frame
for call for parameters and local variables.

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```



main

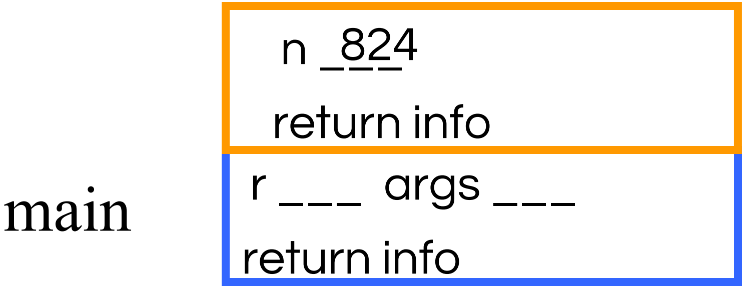
r ___ args ___
return info

main is called. Frame placed on stack.

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```

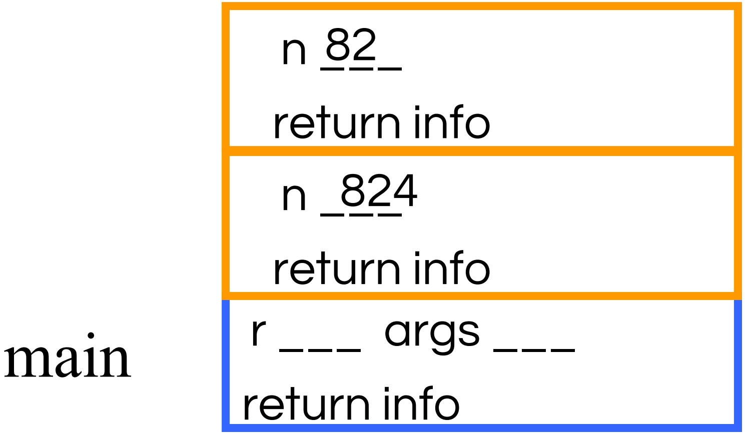


main calls sum with args 824

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```

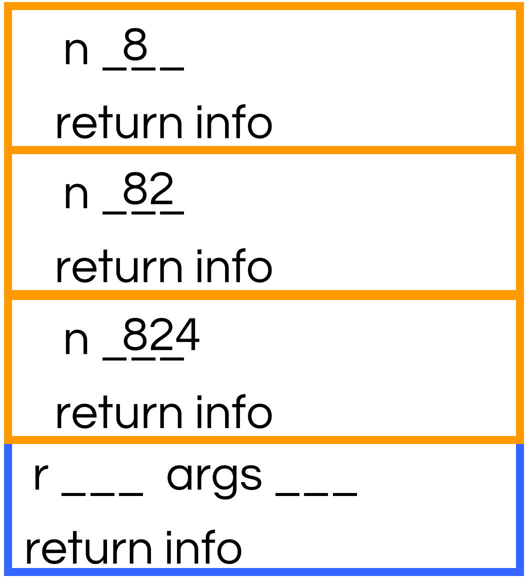


824>=10, sum calls sum

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```



n _8_

return info

n _82_

return info

n _824_

return info

r ___ args ___

return info

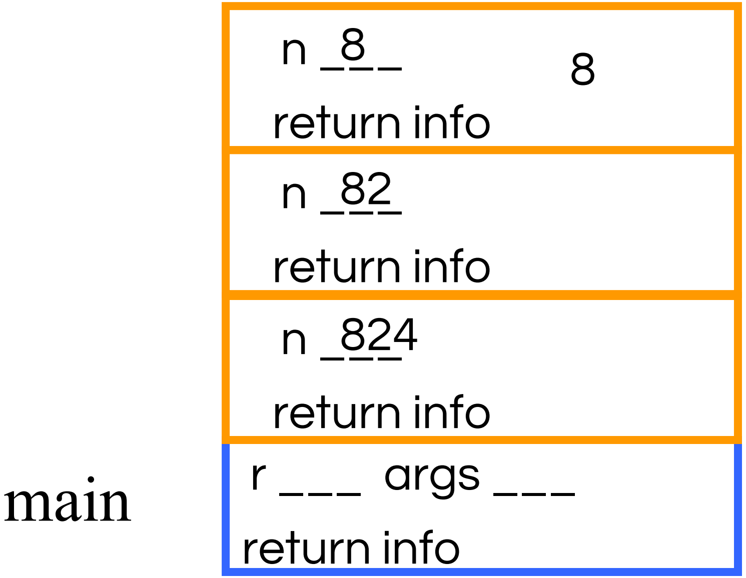
main

82>=10, sum calls sum again

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```

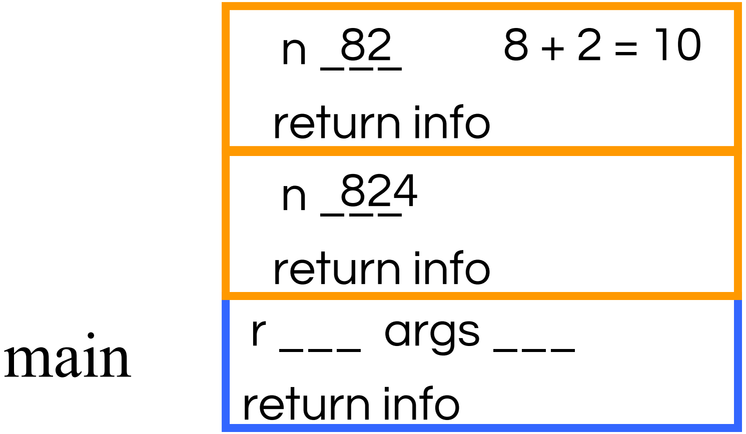

n 8
8
return info
n 82
return info
n 824
return info
main
r ___ args ___
return info

8<10, sum stops: frame is popped and n is put on stack:

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```

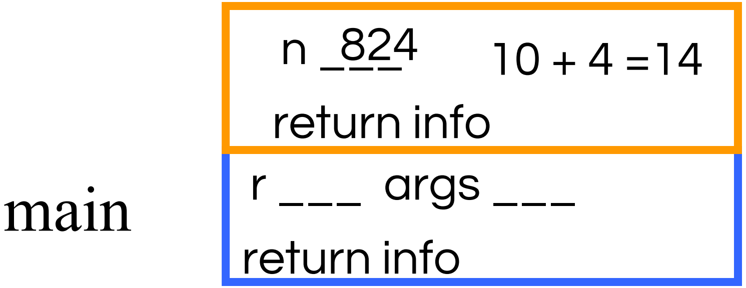


Using return value 8 stack computes
2 + 8 = 10 pops frame from stack puts
return value 10 on stack:

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```

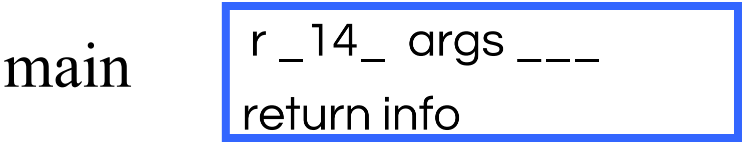


Using return value 10 stack computes
4 + 10 = 14 pops frame from stack puts
return value 14 on stack

An implementation detour: implementation of sum

```
public static int sum(int n) {
     if (n < 10) return n;
     return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```



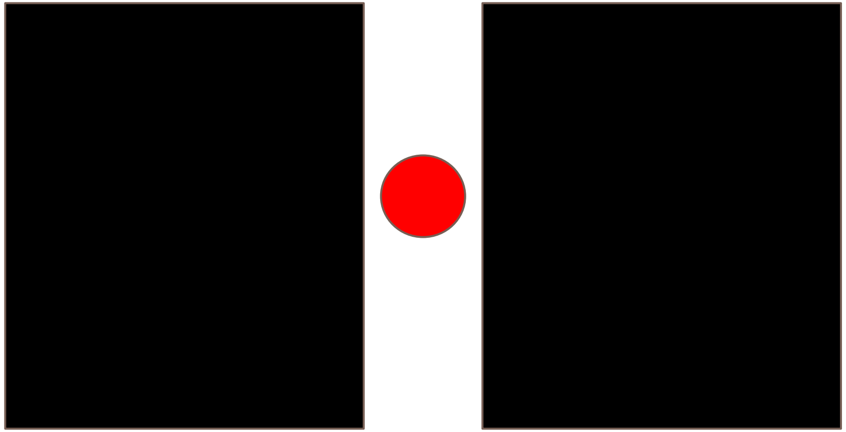
Using return value 14 main stores
14 in r and removes 14 from stack

What do these methods have in common?

- List my ancestors
  - ancestor = parents U ancestor(parents)
- Factorials
  - 0! = 1 n! = n x (n-1)!
- Power of a number
  - b^0 = 1 ;
  - b^c = b * b ^ (c-1)

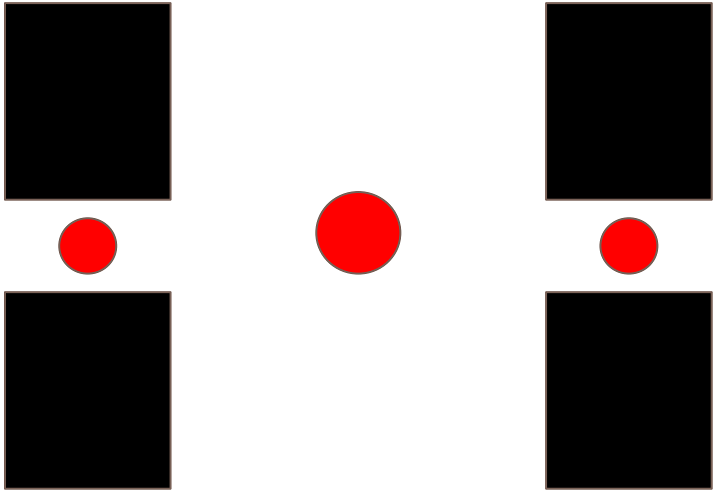
What do these methods have in common?

- List my ancestors
  - ancestor = parents U ancestor(parents)
- Factorials
  - 0! = 1 n! = n x (n-1)!
- Power of a number
  - b^0 = 1 ;
  - b^c = b * b ^ (c-1)



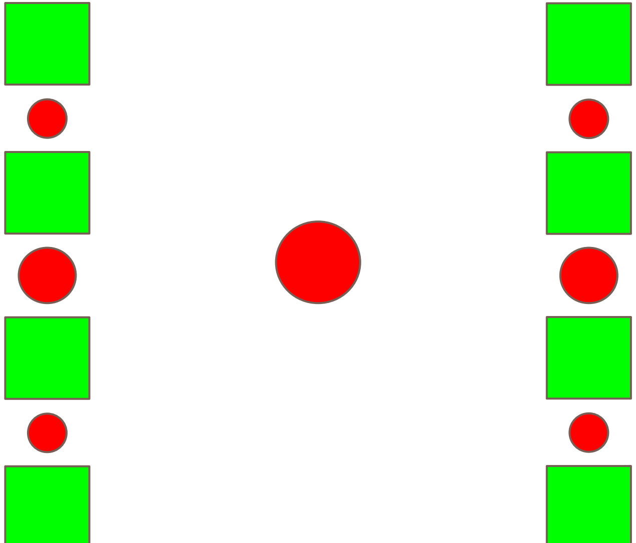
What do these methods have in common?

- List my ancestors
  - ancestor = parents U ancestor(parents)
- Factorials
  - 0! = 1 n! = n x (n-1)!
- Power of a number
  - b^0 = 1 ;
  - b^c = b * b ^ (c-1)



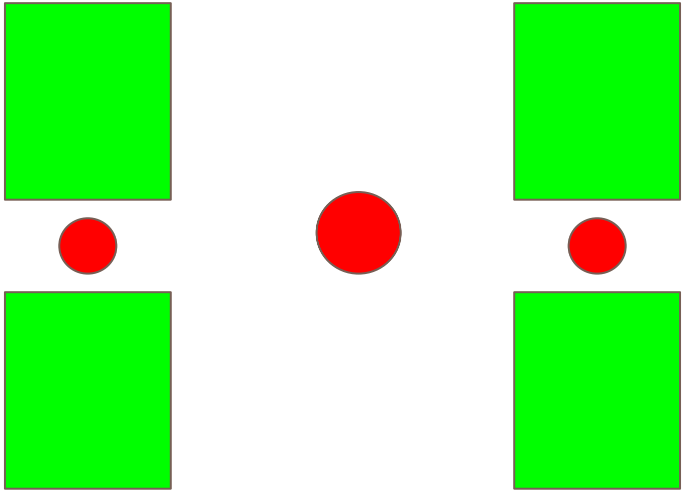
What do these methods have in common?

- List my ancestors
  - ancestor = parents U ancestor(parents)
- Factorials
  - 0! = 1 n! = n x (n-1)!
- Power of a number
  - b^0 = 1 ;
  - b^c = b * b ^ (c-1)



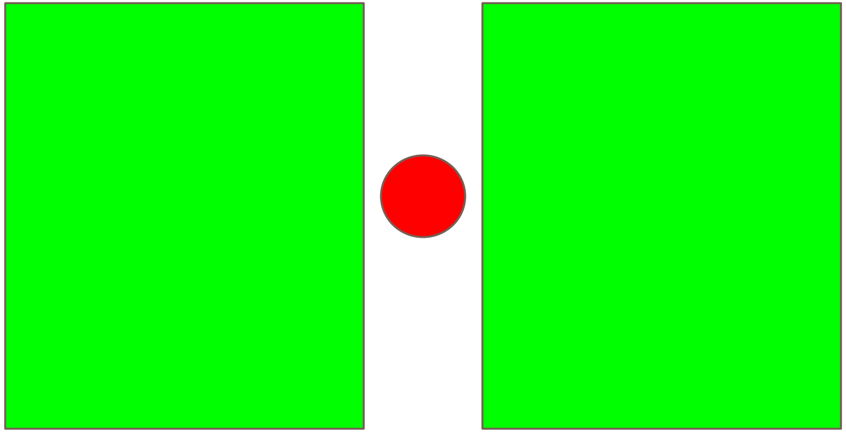
What do these methods have in common?

- List my ancestors
  - ancestor = parents U ancestor(parents)
- Factorials
  - 0! = 1 n! = n x (n-1)!
- Power of a number
  - b^0 = 1 ;
  - b^c = b * b ^ (c-1)



What do these methods have in common?

- List my ancestors
  - ancestor = parents U ancestor(parents)
- Factorials
  - 0! = 1 n! = n x (n-1)!
- Power of a number
  - b^0 = 1 ;
  - b^c = b * b ^ (c-1)



What do these methods have in common?

- □ List my ancestors
  - □ ancestor = parents U ancestor(parents)
- □ Factorials
  - □ 0! = 1 n! = n x (n-1)!
- □ Power of a number
  - □ b^0 = 1 ;
  - □ b^c = b * b ^ (c-1)

What do these methods have in common?

- Every recursive method has one (or more) **base cases**
  - Scenarios that we can solve easily without recursion
  - **0! = 1 .. b^0**
- Every recursive method has one (or more) **recursive cases**
  - The function calls itself on inputs that are “closer” to the base case
  - n! = n x (n-1)! .. b^c = b * b ^ (c-1)
- Every recursive method “**merges**” the result of each recursive call
  - n! = n **x** (n-1)! .. b^c = b * b **^** (c-1)

To design a recursive method

- Write a precise spec
  - Spec of sum(n) says the value of a call equals the sum of the digits of n
- Identify a **base case**, and check, with concrete numbers that the method returns correct values in the base case
  - Ex: sum(8) = 8
- Look at the **recursive case(s)**. In your mind replace each recursive call by what it does acc. to the spec and verify correctness.
  - Ex: sum(82) = 2 + sum(8)
- (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the args of the method.

Powers and Factorials

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

```
/** Computes n!
Precondition n>=0 **/
public static int fact(int n) {
   if (n == 0) return 1;
   return n * fact(n-1);
}
```

- Power of a number
  - b^0 = 1 ;
  - b^c = b * b ^ (c-1)

```
public static int exp(int b, int c) {
   if (c == 0) return 1;
   return b * exp(b, c-1);
}
```

Counting all the letters e in a string

- Return the number of times that the letter e appears in a string using recursion.

ex: countE(“natacha”) = 0
countE(“e”) = 1
countE(“elephant”) = 2

- What do you think is the base case?
  - Think of the smallest possible string

- What about the recursive call? How do we **merge** multiple calls?

Counting all the letters e in a string

- Return the number of times that the letter e appears in a string using recursion.

ex: countE(“natacha”) = 0
countE(“e”) = 1
countE(“elephant”) = 2

- What do you think is the base case?
    - Think of the smallest possible string

- What about the recursive call? How do we **merge** multiple calls?

```
 /** Number of times e occurs in str **/
public static int countX(String str) {
    if (str.equals("")) return 0;
    else {
      char c = str.charAt(0);
      if (c == 'e') {
        return 1 + countX(str.substring(1));
      }
      else return 0 + countX(str.substring(1));
    }
  }
```

Fibonacci Numbers

Mathematical definition:

fib(0) = 0
fib(1) = 1
fib(n) = fib(n – 1) + fib(n – 2) $n \geq 2$

Fibonacci sequence: 0 1 1 2 3 5 8 13 …

Fruit sprouts of a pineapple
Flowering of an artichoke
Fibonacci Heaps
Fibonacci Cubes (ways to organise distributed systems together)
Applications in chemical graph theory…

Fibonacci Numbers

Mathematical definition:

$\text{fib}(0) = 0$

$\text{fib}(1) = 1$

$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2) \quad n \geq 2$

← two base cases!

Fibonacci sequence: 0 1 1 2 3 5 8 13 …

```
/** = fibonacci(n). Pre: n >= 0 */
static int fib(int n) {
   // fib(0) = 1, fib(1) = 2
   if (n <= 1) return n;
   return fib(n-1) + fib(n-2);
}
```

Fruit sprouts of a pineapple
Flowering of an artichoke
Fibonacci Heaps
Fibonacci Cubes (ways to organise distributed systems together)
Applications in chemical graph theory…

Palindromes

- A palindrome is a string that reads the same backward and forwards
  - isPal(racecar) = true isPal(pumpkin) = false isPal(n) = true

Palindromes

- A palindrome is a string that reads the same backward and forwards
    - isPal(racecar) = true   isPal(pumpkin) = false   isPal(n) = true

- How do we go about implementing this?
    - Let's try to rephrase it as a recursive definition

Palindromes

- A palindrome is a string that reads the same backward and forwards
  - isPal(racecar) = true isPal(pumpkin) = false isPal(n) = true

- How do we go about implementing this?
  - Let’s try to rephrase it as a recursive definition

- A String with at least two characters is a palindrome if
  - its first and last characters are equal and
  - chars between first & last characters are also palindrome:

Palindromes

- A palindrome is a string that reads the same backward and forwards
  - isPal(racecar) = true isPal(pumpkin) = false isPal(n) = true
- How do we go about implementing this?
  - Let's try to rephrase it as a recursive definition
- A String with at least two characters is a palindrome if
  - its first and last characters are equal and
  - chars between first & last characters are also palindrome:

What are we missing here?

Palindromes

- A palindrome is a string that reads the same backward and forwards
  - isPal(racecar) = true   isPal(pumpkin) = false   isPal(n) = true

- How do we go about implementing this?
  - Let's try to rephrase it as a recursive definition

- A String with at least two characters is a palindrome if
  - It is of length 0/1 **or**
  - its first and last characters are equal **and**
  - chars between first & last characters are also palindrome:

Basecase!

Palindromes

```
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1)
        return true;

    // { s has at least 2 chars }
    int n= s.length()-1;
    return s.charAt(0) == s.charAt(n)  &&  isPal(s.substring(1,n));
}
```

Substring from s[1] to s[n-1]

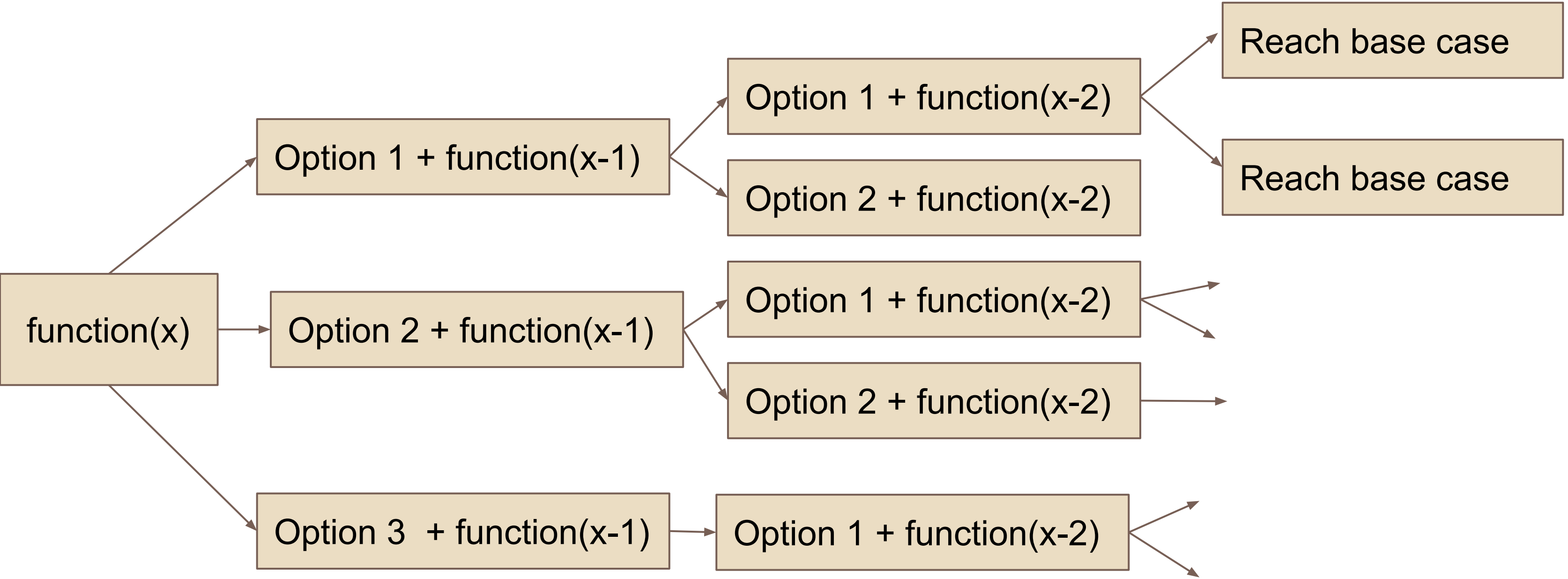
Recursion with backtracking

- Some recursion problems require you to **enumerate all solutions (Type 1)**

- Find solutions **subject to a set of constraints (Type 2)**

- Follow a similar format:
  - Explore one solution until the end:
    - If Type 1: add that solution to a set, and backtrack to the last recursive call to explore other solutions
    - If Type 2: if solution satisfies constraint, return true, otherwise, backtrack to the last recursive call to explore other solutions

Recursion with backtracking



I am writing x-1 to mean “smaller input” and + to mean “merge”

Finding permutations of a string

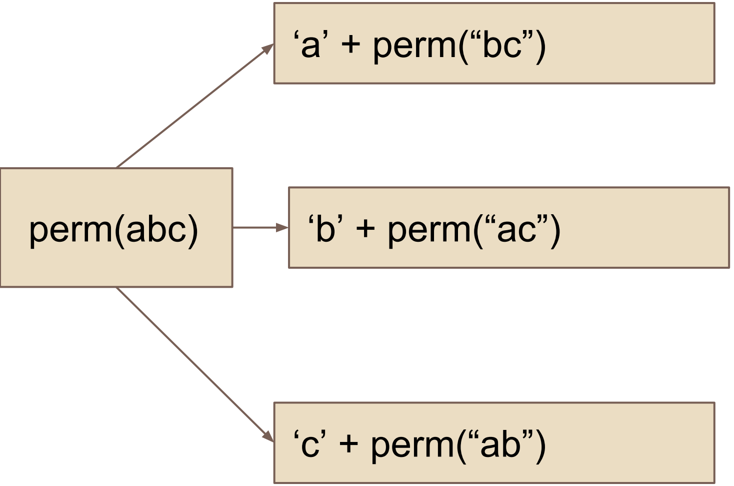
- Compute all permutations of a string (assuming the string has distinct characters)
  - perms(abc) = abc, acb, bac, bca, cab, cba

- Recursive Definition
  - Each possible first letter, followed by all permutations of the remaining characters

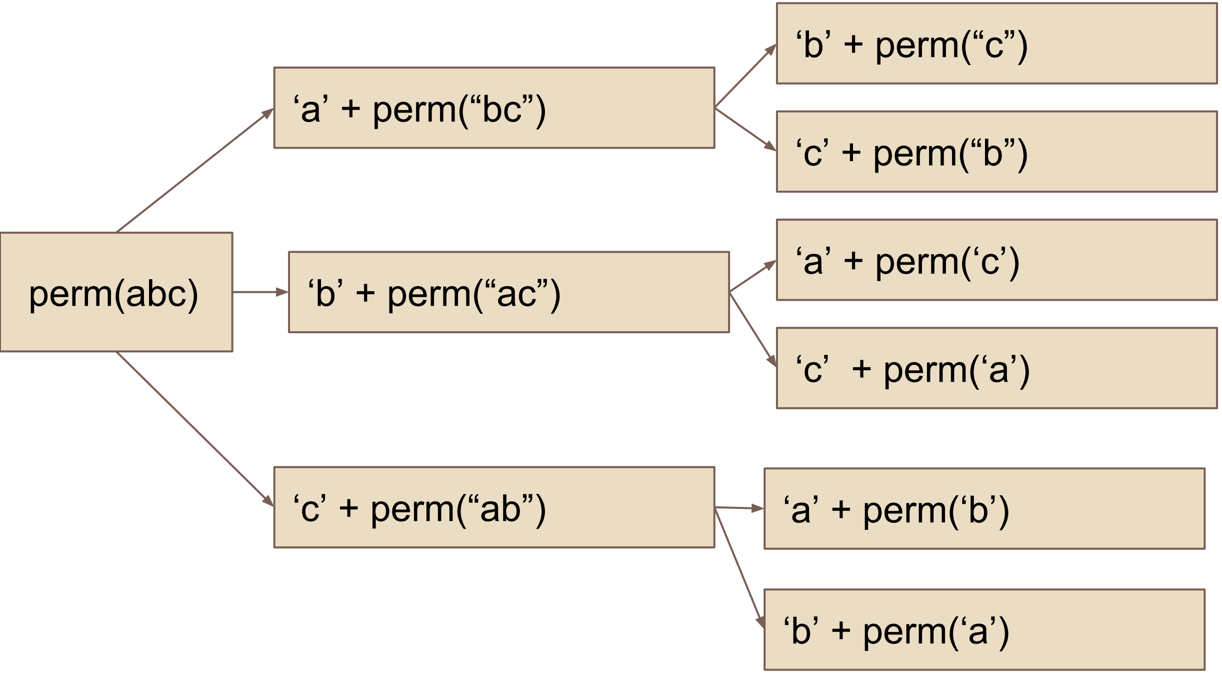
Finding permutations of a string

perm(abc)

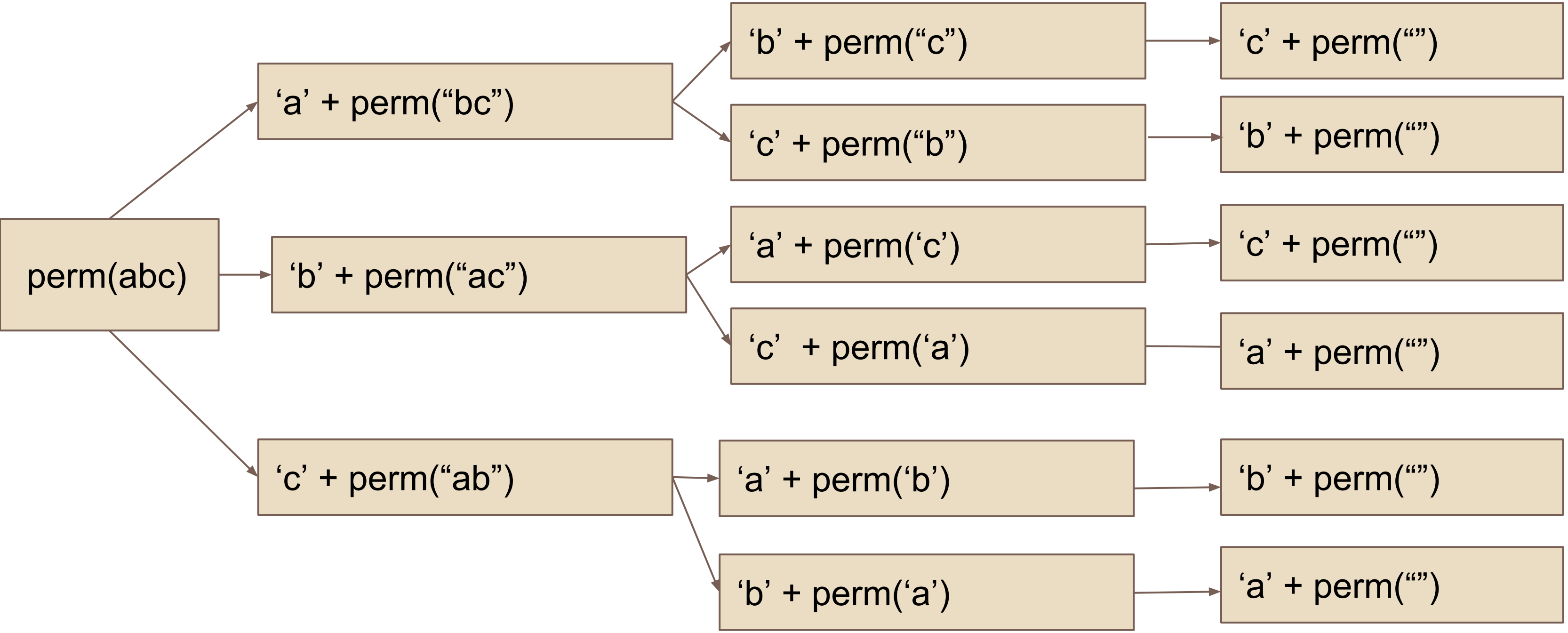
Finding permutations of a string



Finding permutations of a string



Finding permutations of a string



perm(abc)

‘a’ + perm(“bc”)

‘b’ + perm(“c”)

‘c’ + perm(“”)

‘c’ + perm(“b”)

‘b’ + perm(“”)

‘b’ + perm(“ac”)

‘a’ + perm(‘c’)

‘c’ + perm(“”)

‘c’ + perm(‘a’)

‘a’ + perm(“”)

‘c’ + perm(“ab”)

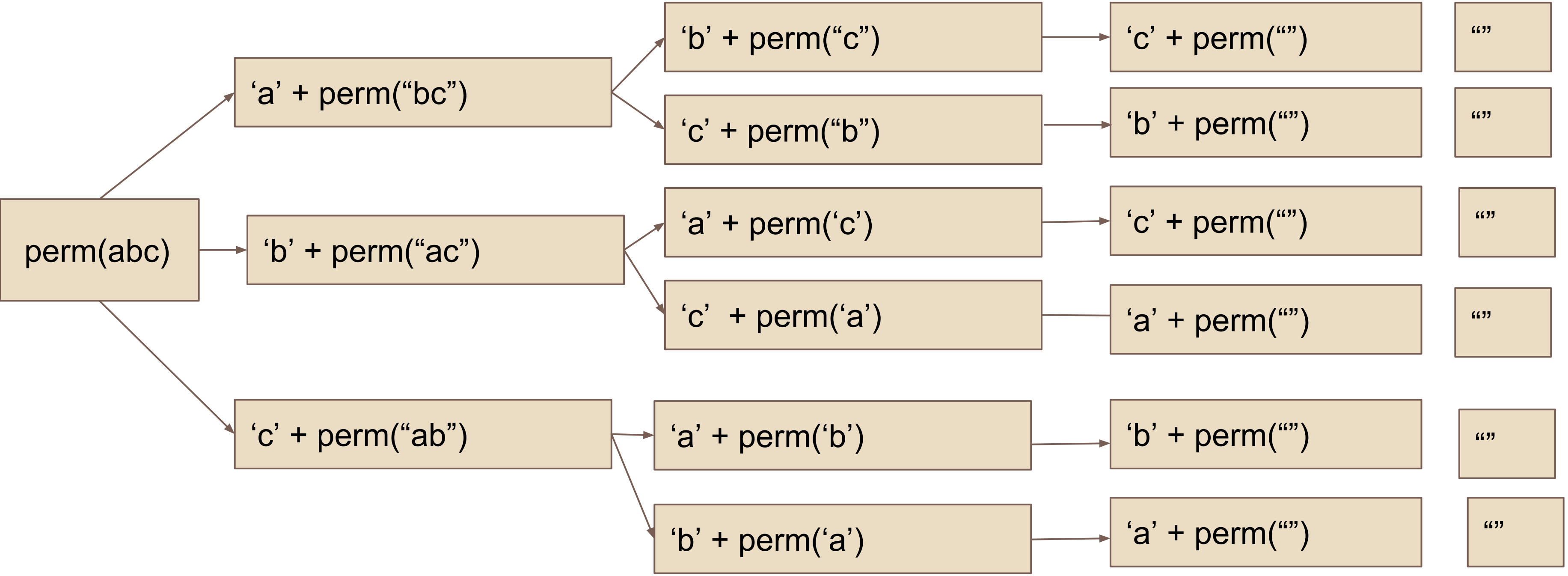
‘a’ + perm(‘b’)

‘b’ + perm(“”)

‘b’ + perm(‘a’)

‘a’ + perm(“”)

Finding permutations of a string



perm(abc)

‘a’ + perm(“bc”)

‘b’ + perm(“c”)

‘c’ + perm(“”)

“”

‘c’ + perm(“b”)

‘b’ + perm(“”)

“”

‘b’ + perm(“ac”)

‘a’ + perm(‘c’)

‘c’ + perm(“”)

“”

‘c’ + perm(‘a’)

‘a’ + perm(“”)

“”

‘c’ + perm(“ab”)

‘a’ + perm(‘b’)

‘b’ + perm(“”)

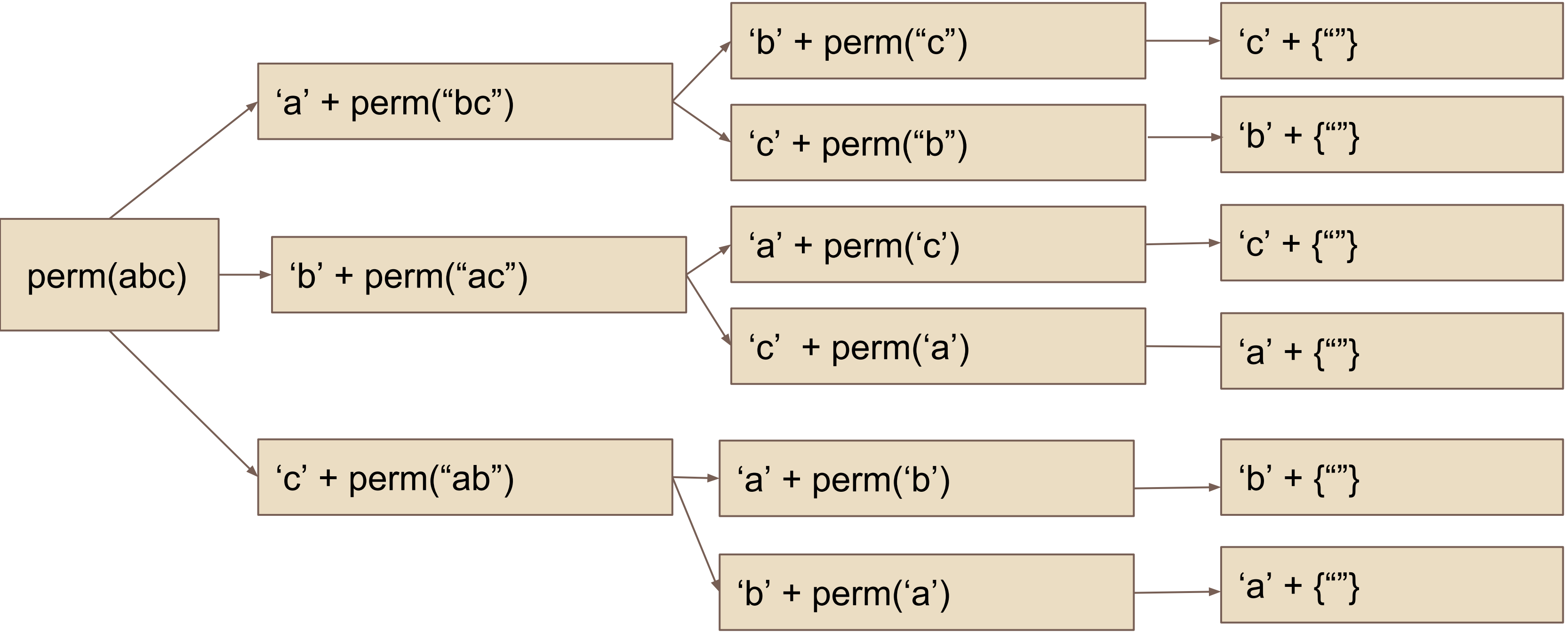
“”

‘b’ + perm(‘a’)

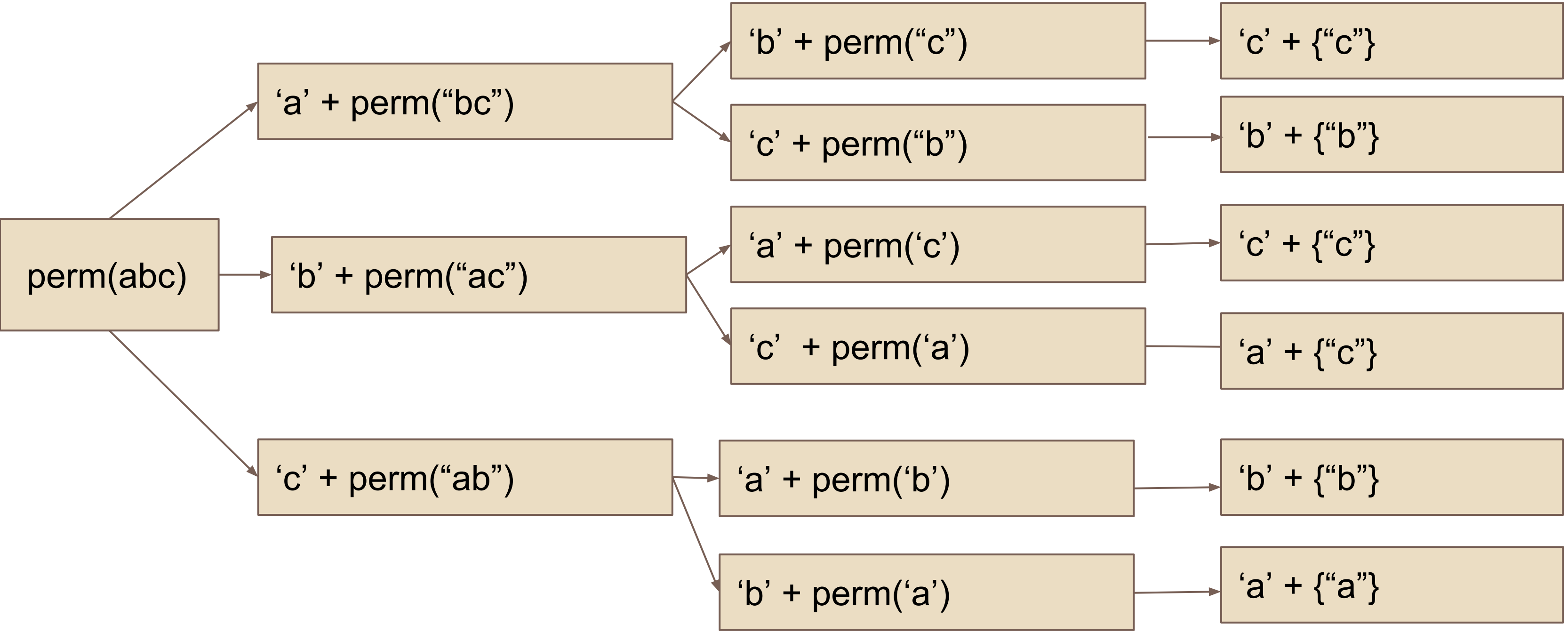
‘a’ + perm(“”)

“”

Finding permutations of a string



Finding permutations of a string



perm(abc)

‘a’ + perm(“bc”)

‘b’ + perm(“c”)

‘c’ + {“c”}

‘c’ + perm(“b”)

‘b’ + {“b”}

‘b’ + perm(“ac”)

‘a’ + perm(‘c’)

‘c’ + {“c”}

‘c’ + perm(‘a’)

‘a’ + {“c”}

‘c’ + perm(“ab”)

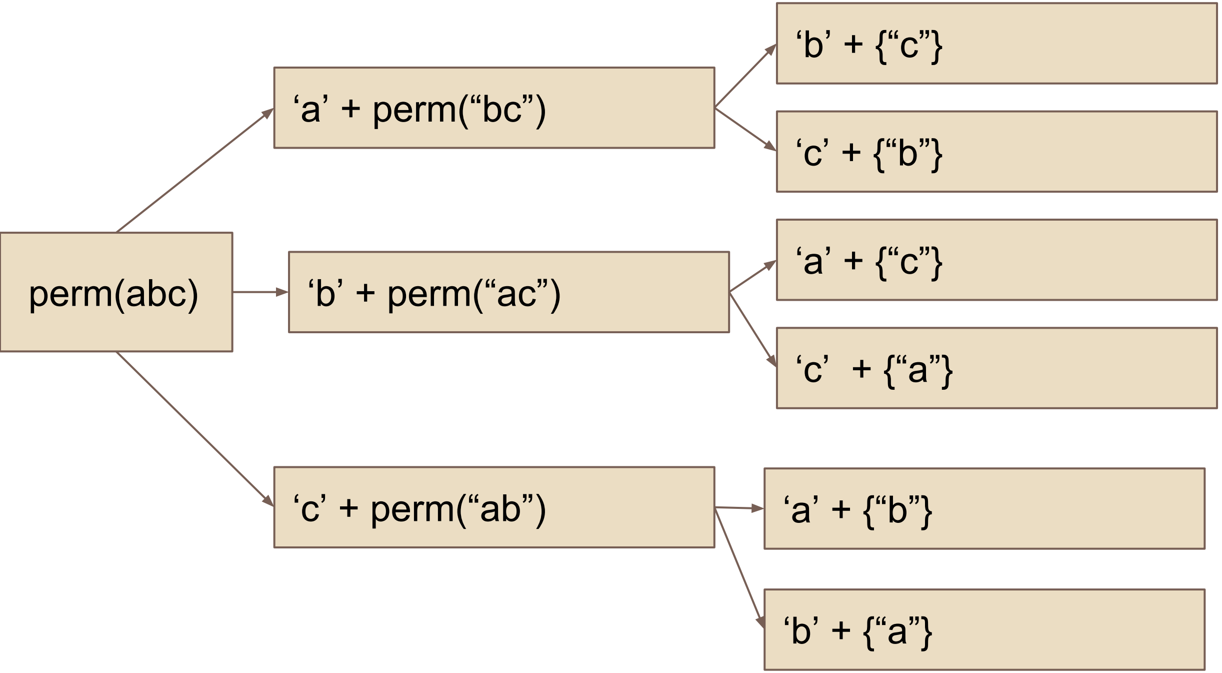
‘a’ + perm(‘b’)

‘b’ + {“b”}

‘b’ + perm(‘a’)

‘a’ + {“a”}

Finding permutations of a string



perm(abc)

‘a’ + perm(“bc”)

‘b’ + {“c”}

‘c’ + {“b”}

‘b’ + perm(“ac”)

‘a’ + {“c”}

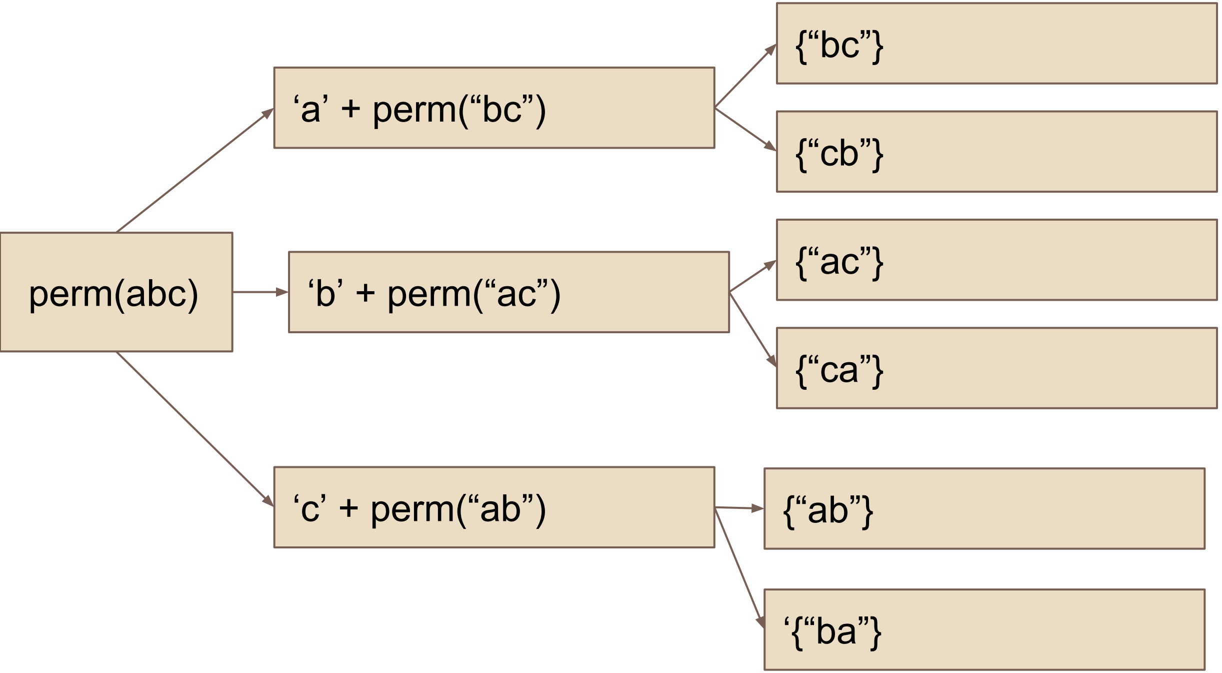
‘c’ + {“a”}

‘c’ + perm(“ab”)

‘a’ + {“b”}

‘b’ + {“a”}

Finding permutations of a string



perm(abc)

‘a’ + perm(“bc”)

‘b’ + perm(“ac”)

‘c’ + perm(“ab”)

{“bc”}

{“cb”}

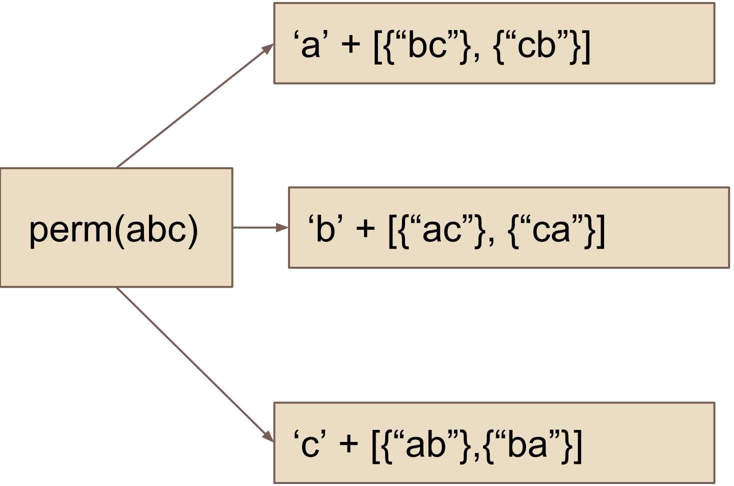
{“ac”}

{“ca”}

{“ab”}

‘{“ba”}

Finding permutations of a string



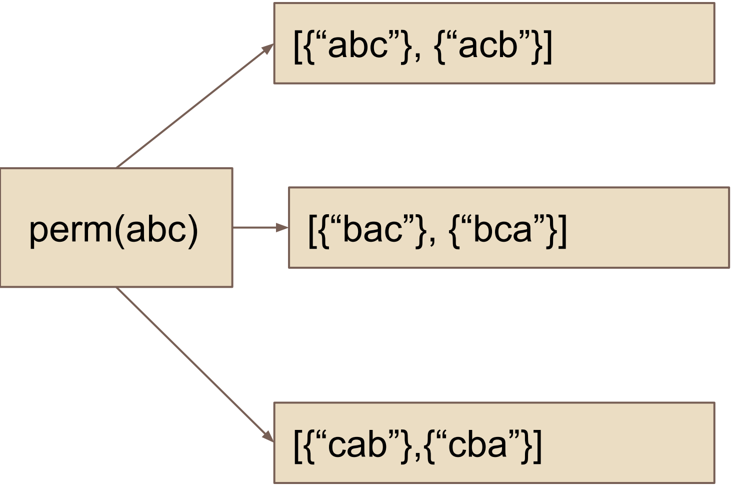
perm(abc)

‘a’ + [{“bc”}, {“cb”}]

‘b’ + [{“ac”}, {“ca”}]

‘c’ + [{“ab”},{“ba”}]

Finding permutations of a string



perm(abc)

[{“abc”}, {“acb”}]

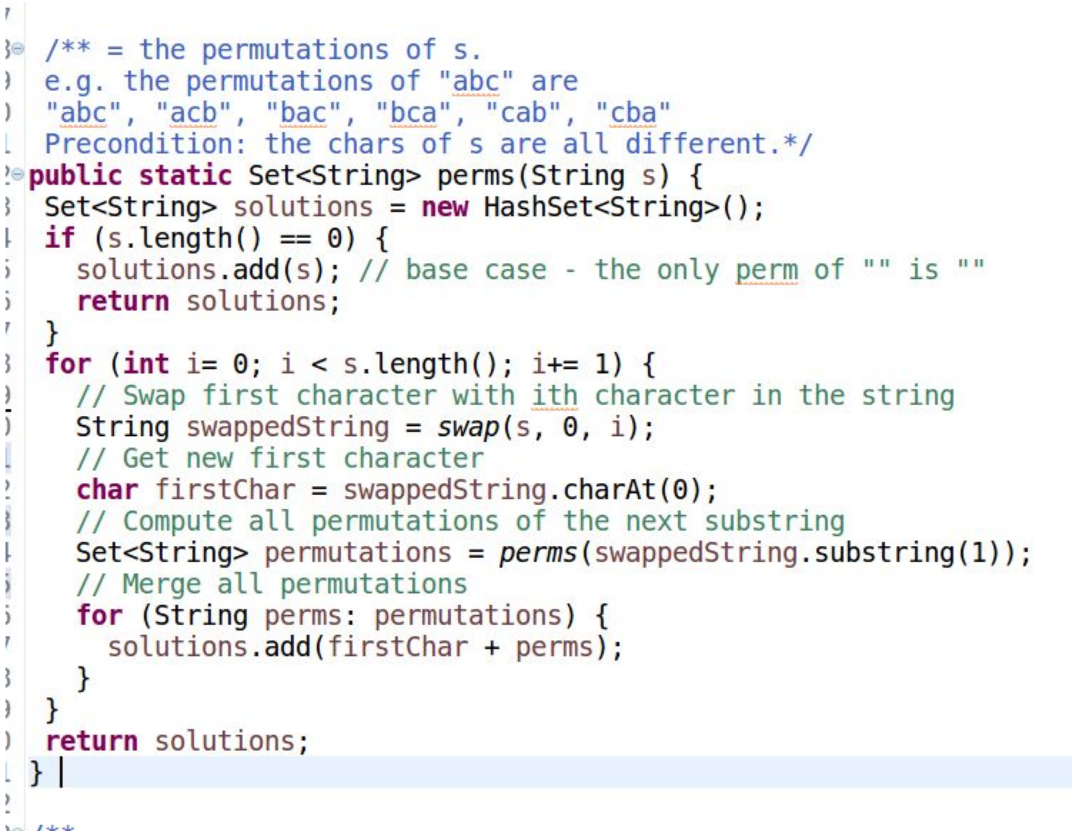
[{“bac”}, {“bca”}]

[{“cab”},{“cba”}]

Finding permutations of a string

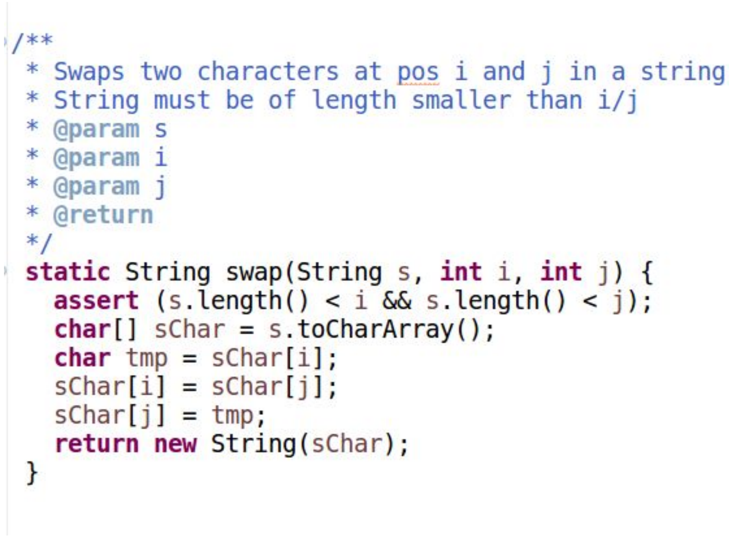
[“abc”,”acb”,”bac”,”bca”,”cab”,”cba”]

Finding permutations of a string



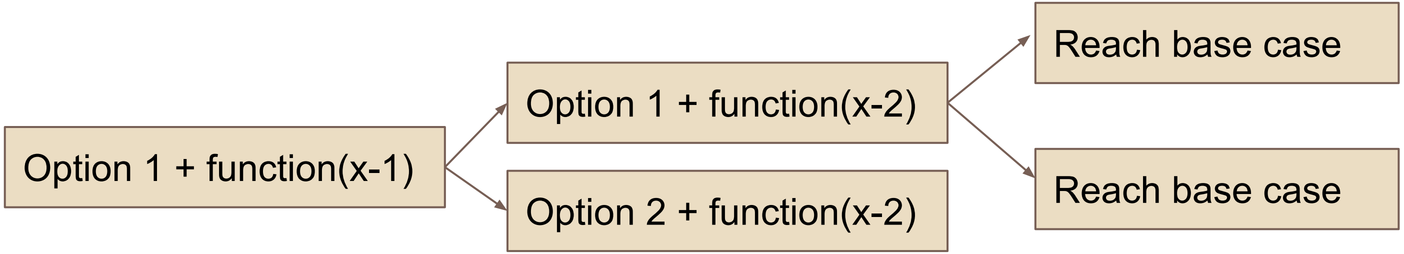
```
/** = the permutations of s.
e.g. the permutations of "abc" are
"abc", "acb", "bac", "bca", "cab", "cba"
Precondition: the chars of s are all different.*/
public static Set<String> perms(String s) {
  Set<String> solutions = new HashSet<String>();
  if (s.length() == 0) {
    solutions.add(s); // base case - the only perm of "" is ""
    return solutions;
  }
  for (int i= 0; i < s.length(); i+= 1) {
    // Swap first character with ith character in the string
    String swappedString = swap(s, 0, i);
    // Get new first character
    char firstChar = swappedString.charAt(0);
    // Compute all permutations of the next substring
    Set<String> permutations = perms(swappedString.substring(1));
    // Merge all permutations
    for (String perms: permutations) {
      solutions.add(firstChar + perms);
    }
  }
  return solutions;
}

/**
```



```
/**
 * Swaps two characters at pos i and j in a string
 * String must be of length smaller than i/j
 * @param s
 * @param i
 * @param j
 * @return
 */
static String swap(String s, int i, int j) {
  assert (s.length() < i && s.length() < j);
  char[] sChar = s.toCharArray();
  char tmp = sChar[i];
  sChar[i] = sChar[j];
  sChar[j] = tmp;
  return new String(sChar);
}
```

Remember:
Recursion with backtracking (Type 1)



Option 1 + function(x-1)

Option 1 + function(x-2)

Option 2 + function(x-2)

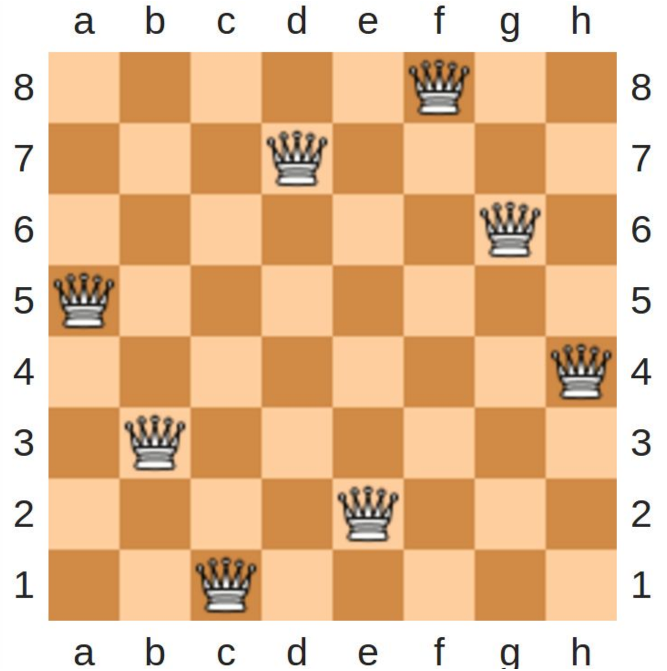
Reach base case

Reach base case

```
function(x) =
    Set<Solutions> set;
    If base case { add x to set }
    else { // Recursive Step
        for every option:
            Set<Solutions> recursive = function(x-1)
            for every solution in recursive:
                merge (option,recursive)
                Add x to set
    }
    Return set;
```

8-queens problem

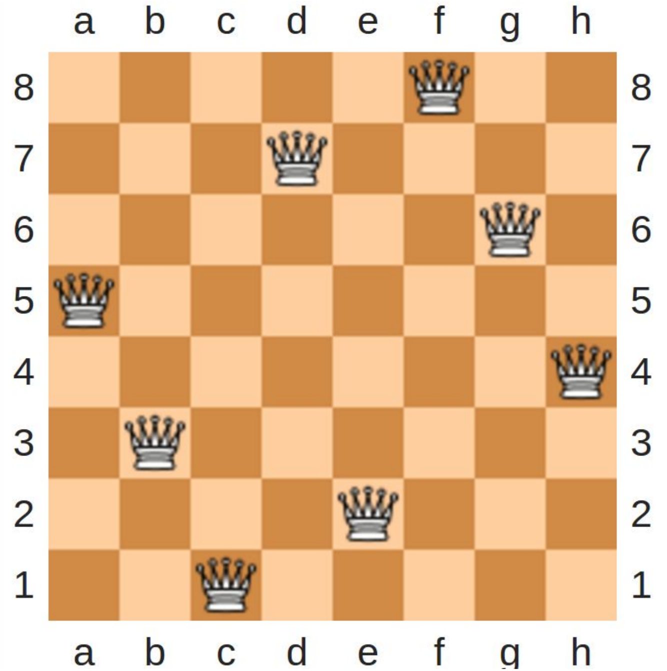
- □ Find the position on an 8 x 8 chessboard such that no queen is attacking the other
  - □ A queen is attacking another queen if they are in the same row, column, or diagonal



8-queens problem

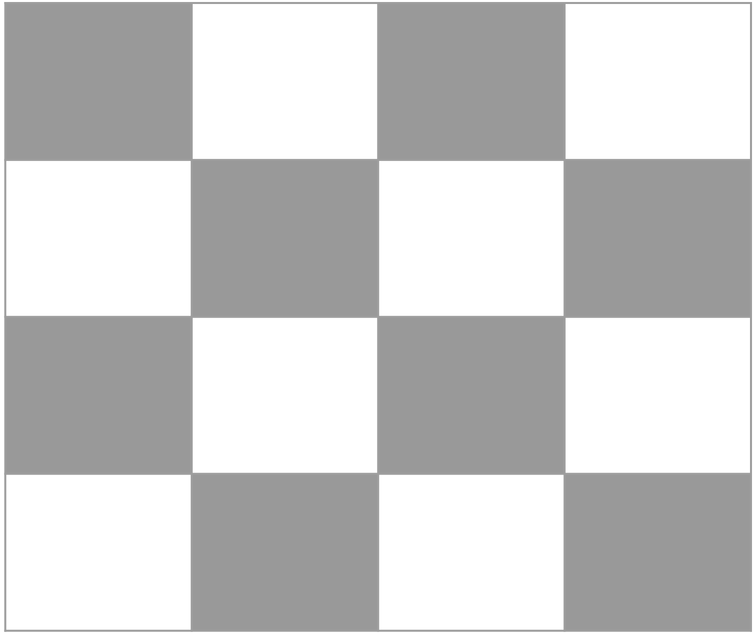
- Find the position on an 8 x 8 chessboard such that no queen is attacking the other
  - A queen is attacking another queen if they are in the same row, column, or diagonal

- Recursive Formulation:
  - Queen 1 is not attacking anyone, and the other n-1 queens solve the n-1-queens problem



8-queens problem

- □ Step 1: solve queen(4, chessboard)



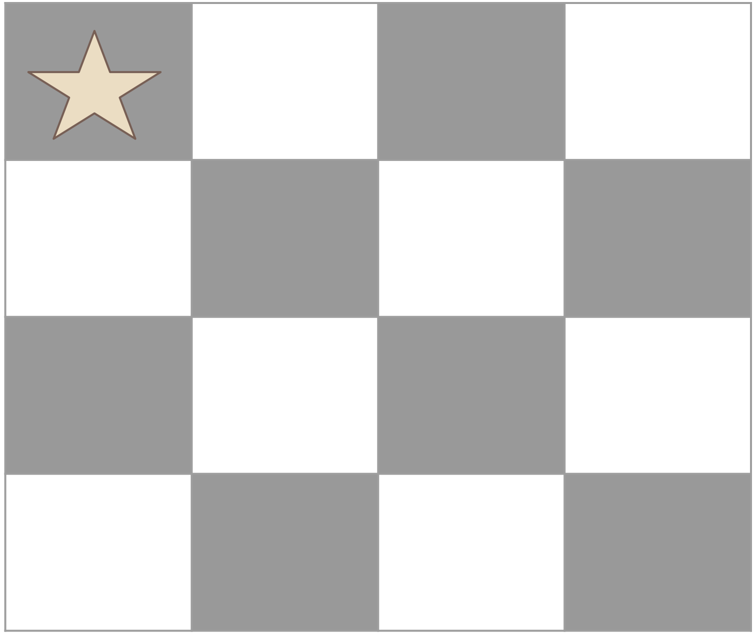
8-queens problem

Frame solveQueen(4, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])




      }
   Return success;
```



Try placing Queen 1 in (0,0)

8-queens problem

Frame solveQueen(4, chess)

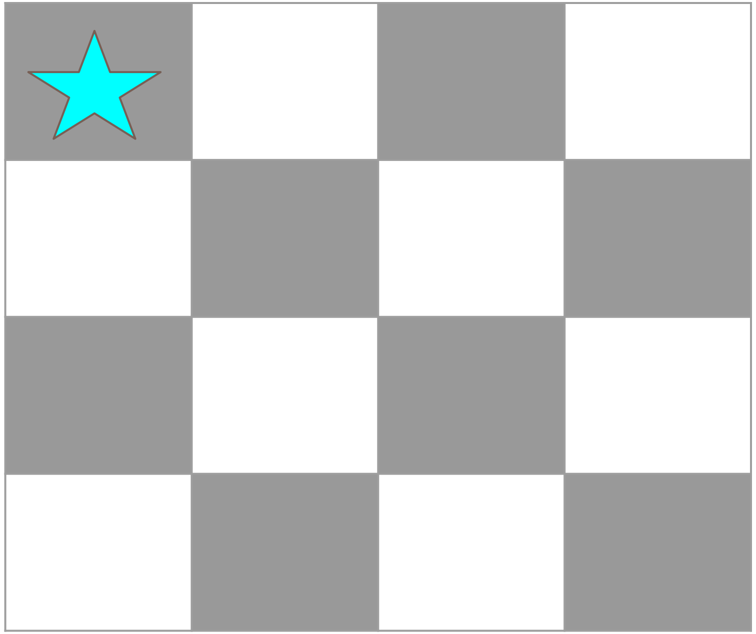
```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])








      }
   Return success;
```



Is queen Safe at (0,0)? Yes!

8-queens problem

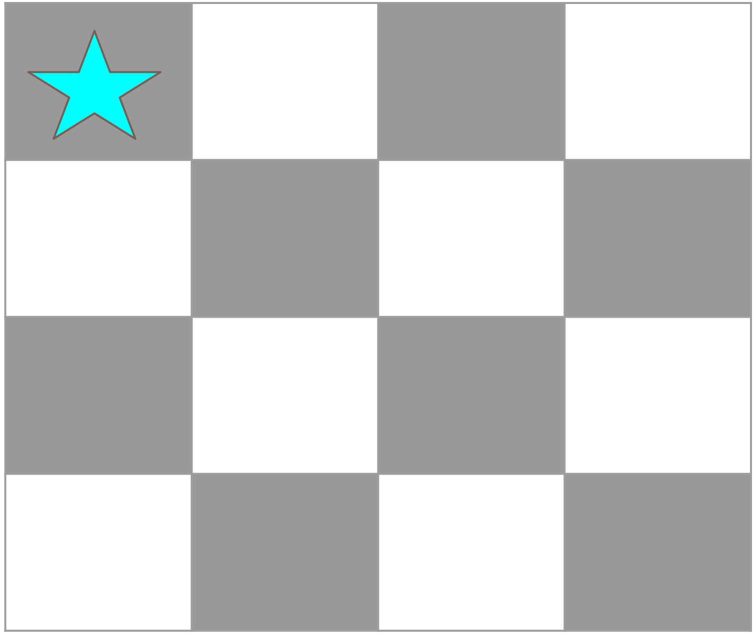
```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;



         }
         }
    Return success;
```

Update chessboard

Frame solveQueen(4, chess)



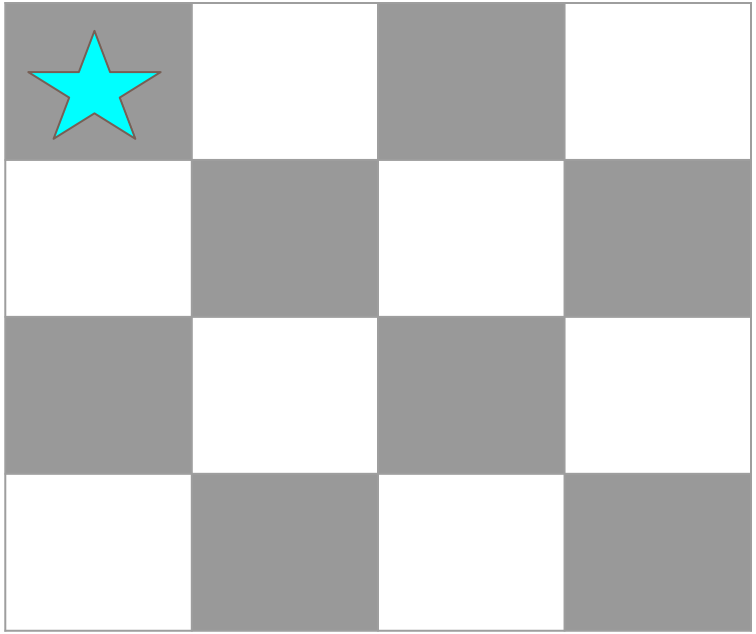
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(3,currentArray);


      }
      }
   Return success;
```

Call solveQueen(3,chess)

Frame solveQueen(4, chess)



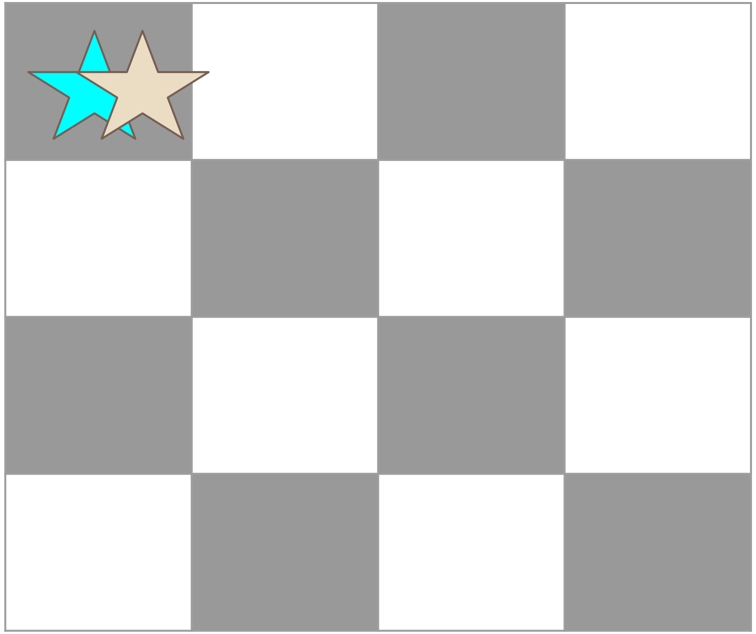
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Try placing Queen 2 in (0,0)

Frame solveQueen(3, chess)



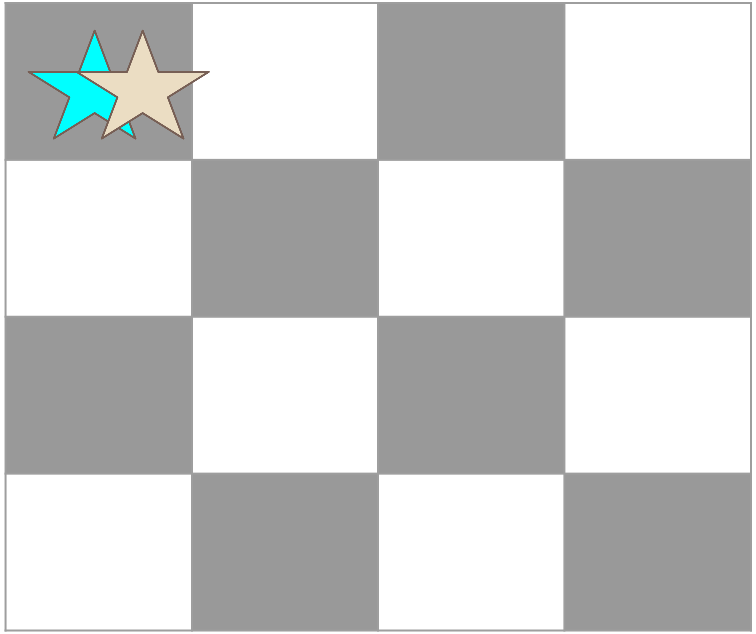
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Is she safe? No

Frame solveQueen(3, chess)



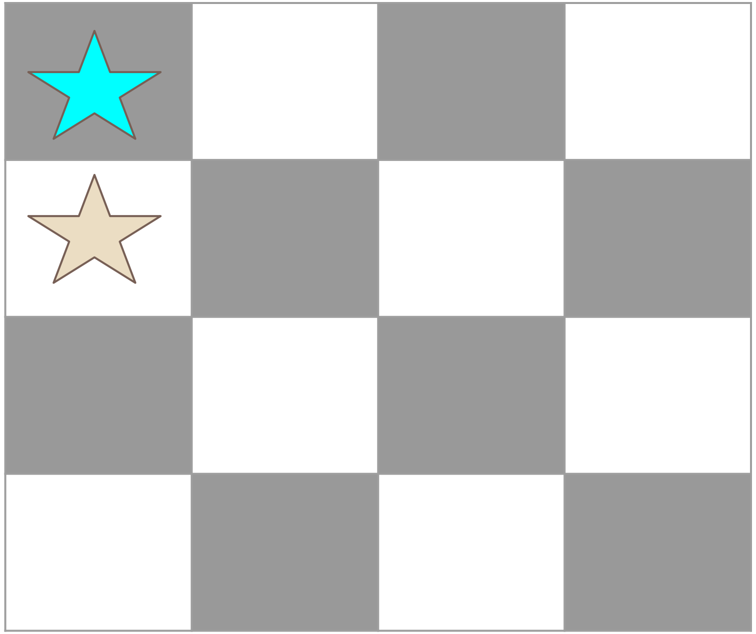
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Try placing Queen 2 in (1,0)

Frame solveQueen(3, chess)



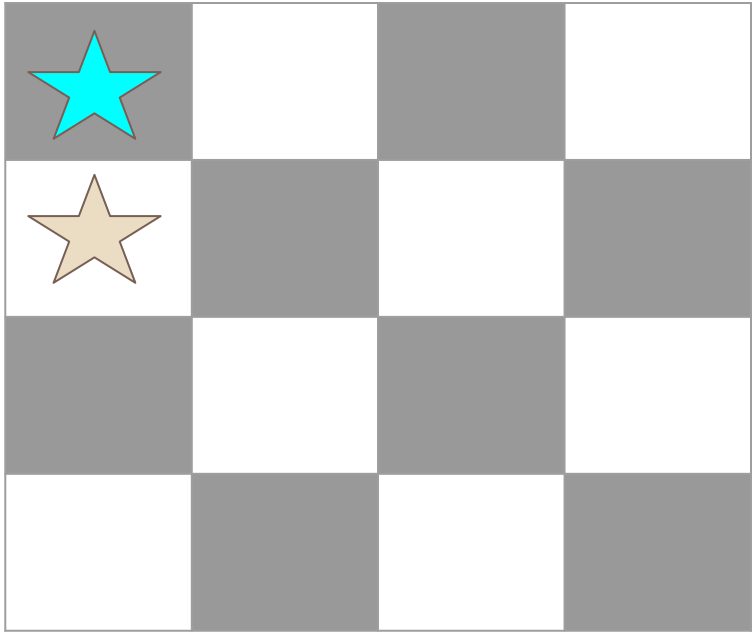
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);

      }
      }
   Return success;
```

Try placing Queen 2 in (0,0)

Frame solveQueen(3, chess)

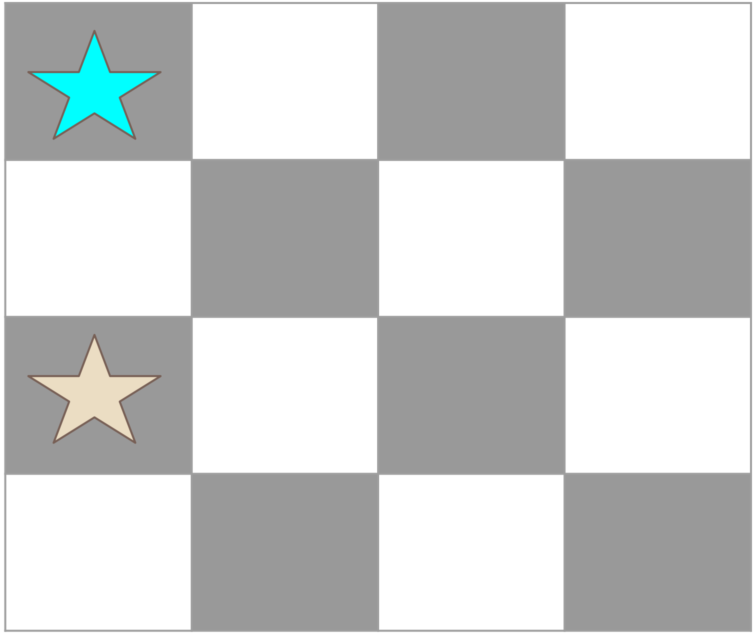


8-queens problem

Frame solveQueen(3, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2currentArray);


      }
      }
  Return success;
```



Try placing Queen 2 in (2,0)

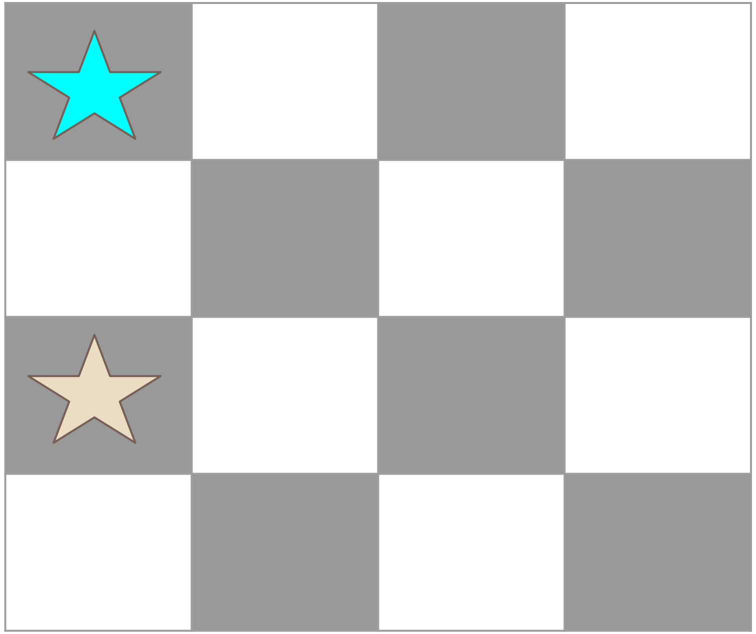
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Is she safe? No

Frame solveQueen(3, chess)

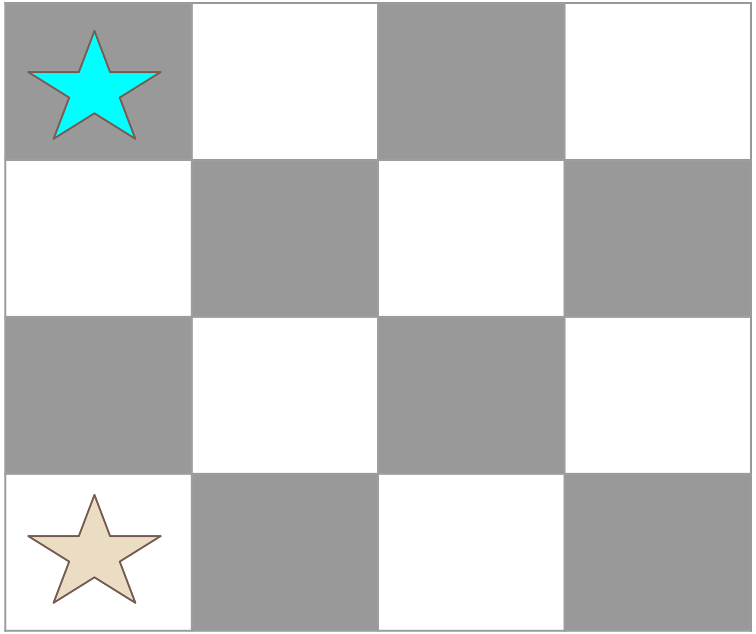


8-queens problem

Frame solveQueen(3, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);

      }
      }
   Return success;
```



Try placing Queen 2 in (3,0)

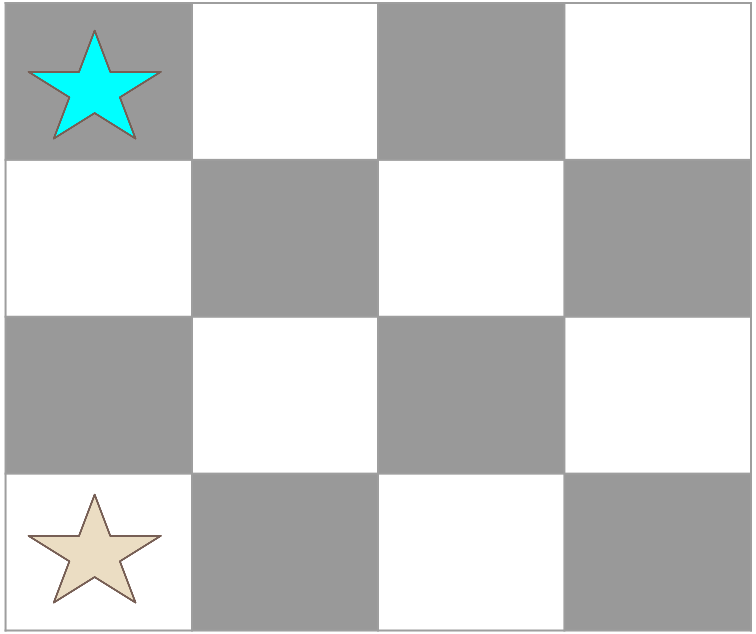
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Is she safe? No

Frame solveQueen(3, chess)



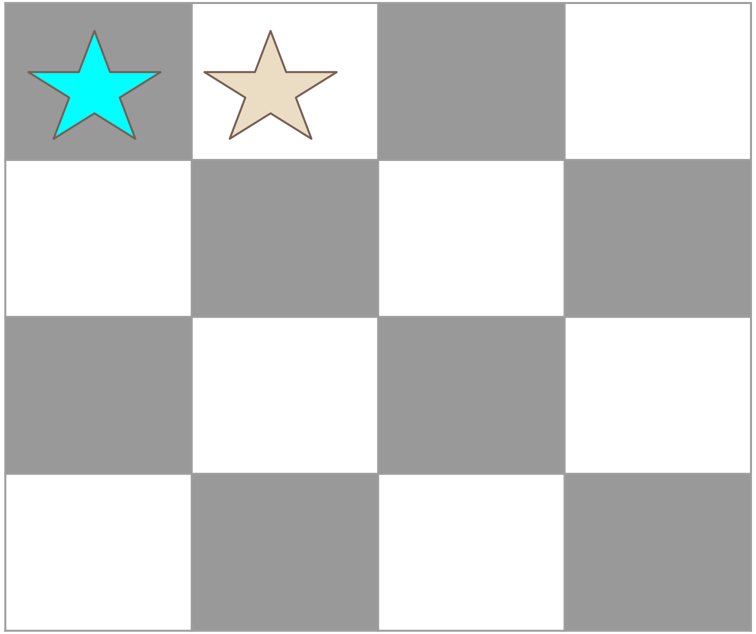
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Try placing Queen 2 in (0,1)

Frame solveQueen(3, chess)

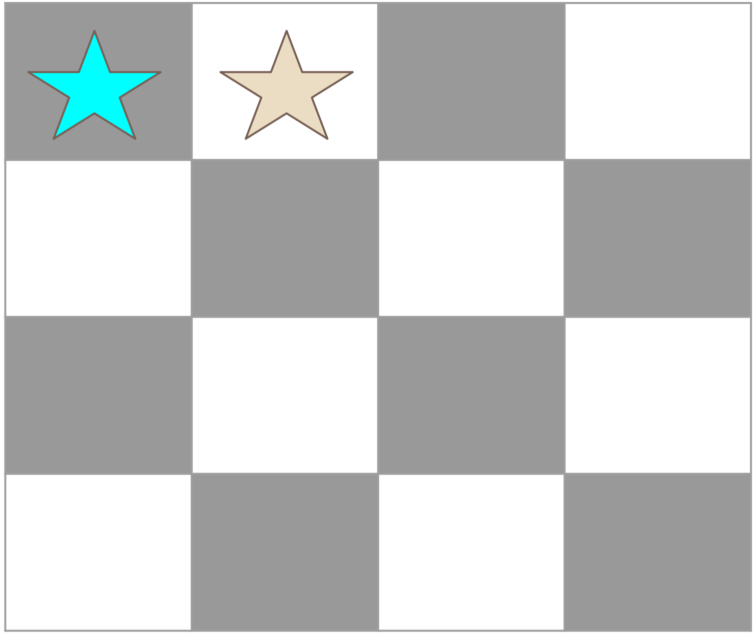


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Frame solveQueen(3, chess)



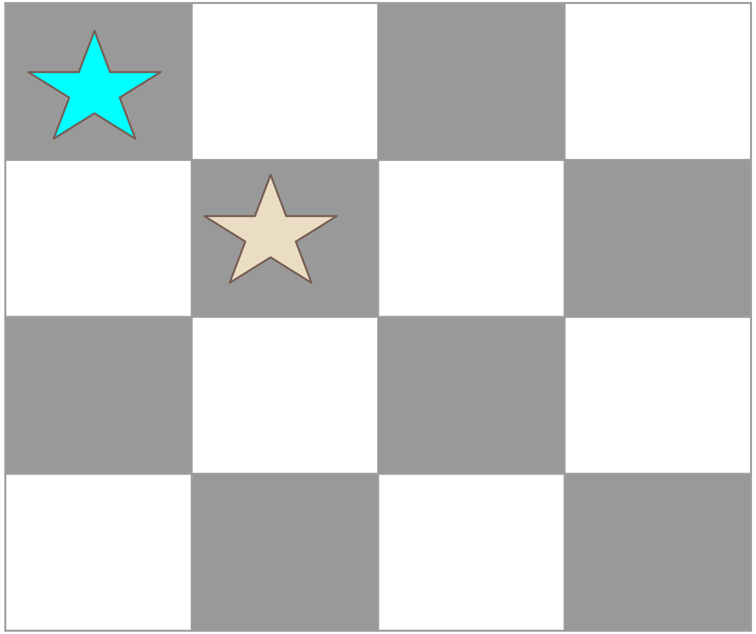
Is she safe? No

8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Frame solveQueen(3, chess)



Try placing Queen 2 in (1,1)

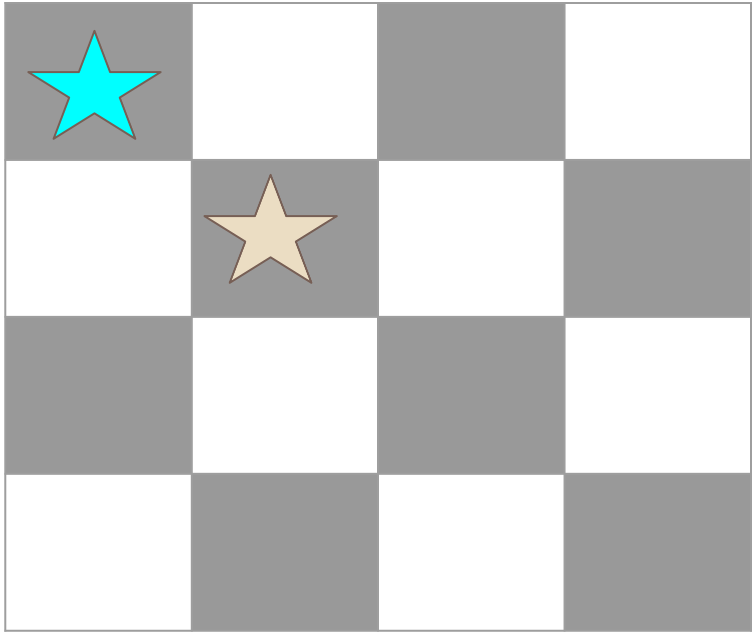
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
  Return success;
```

Is she safe? No

Frame solveQueen(3, chess)



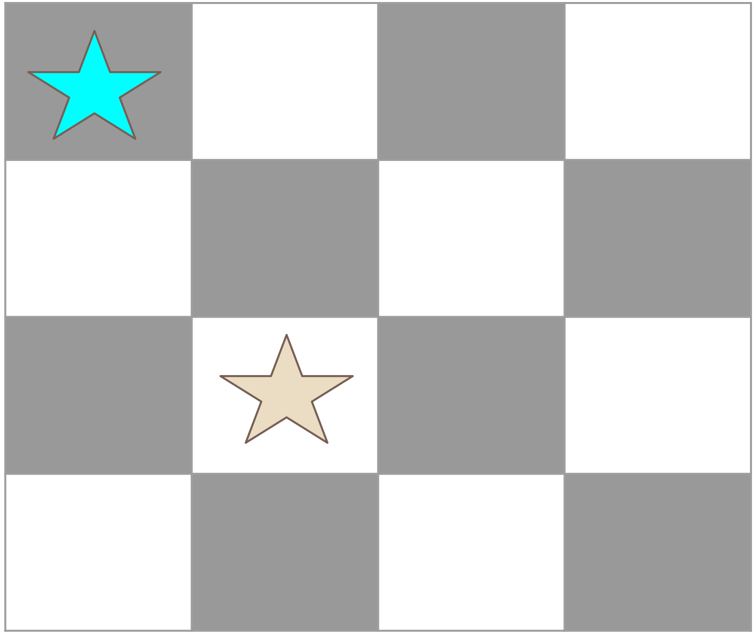
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(2,currentArray);


         }
         }
    Return success;
```

Try placing Queen 2 in (2,1)

Frame solveQueen(3, chess)

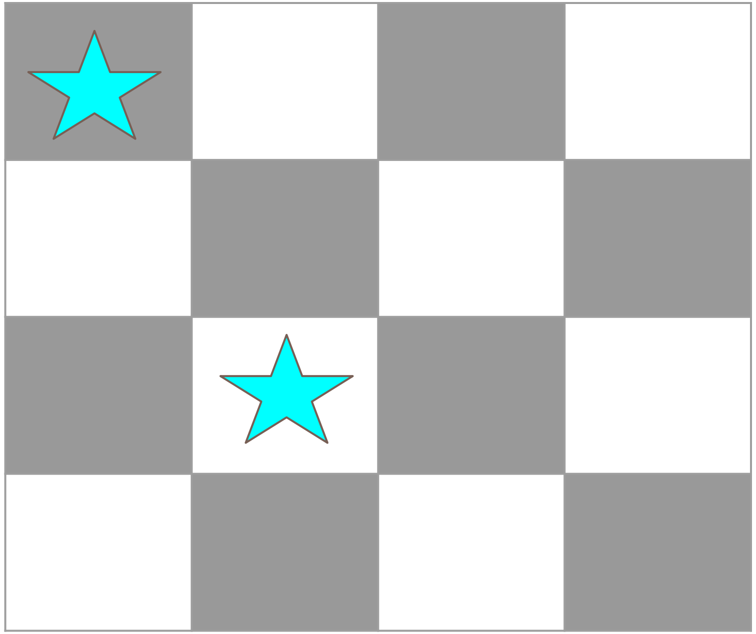


8-queens problem

Frame solveQueen(3, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
        bool success = isSafe(currentArray[row][col])
        If (success) {
           chess[row][col] = 1;
           success  = queen(2,currentArray);

        }
        }
  Return success;
```



Is she safe? Yes!

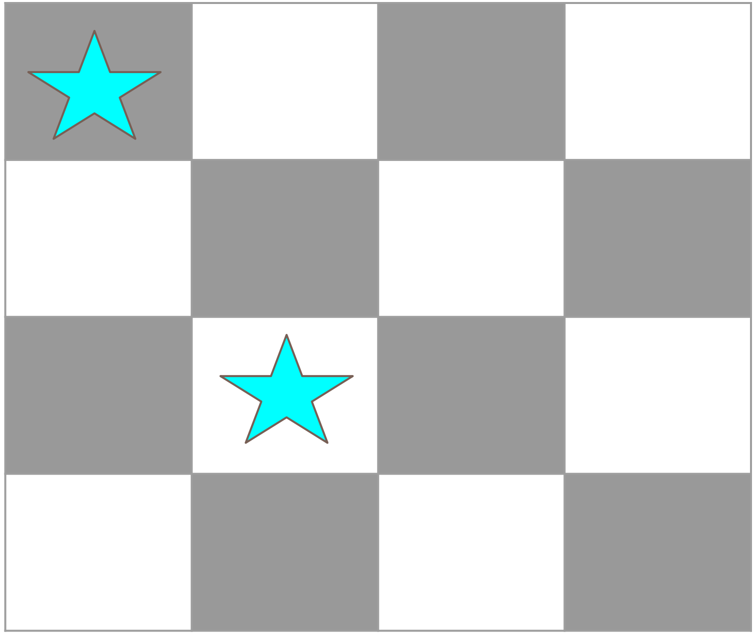
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);

      }
      }
   Return success;
```

Update Chessboard

Frame solveQueen(3, chess)



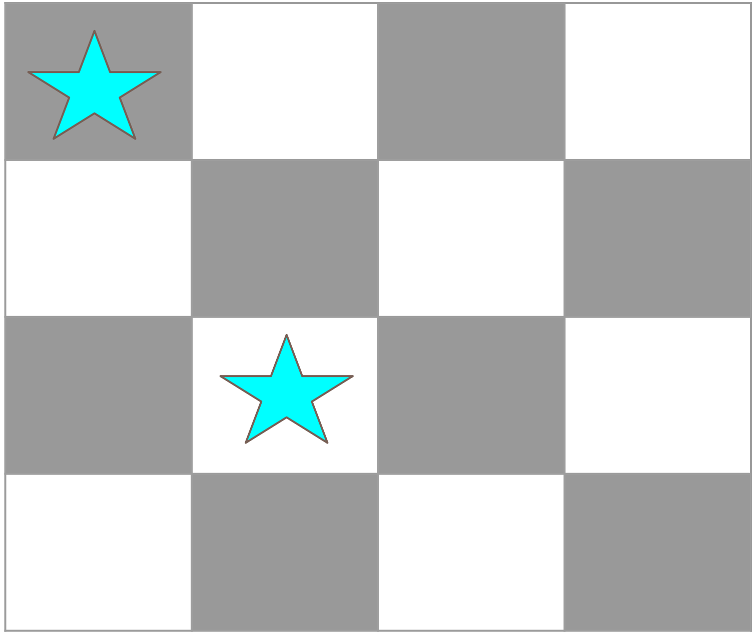
8-queens problem

Frame solveQueen(3, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);


      }
      }
   Return success;
```

Call queen(2,currentArray)



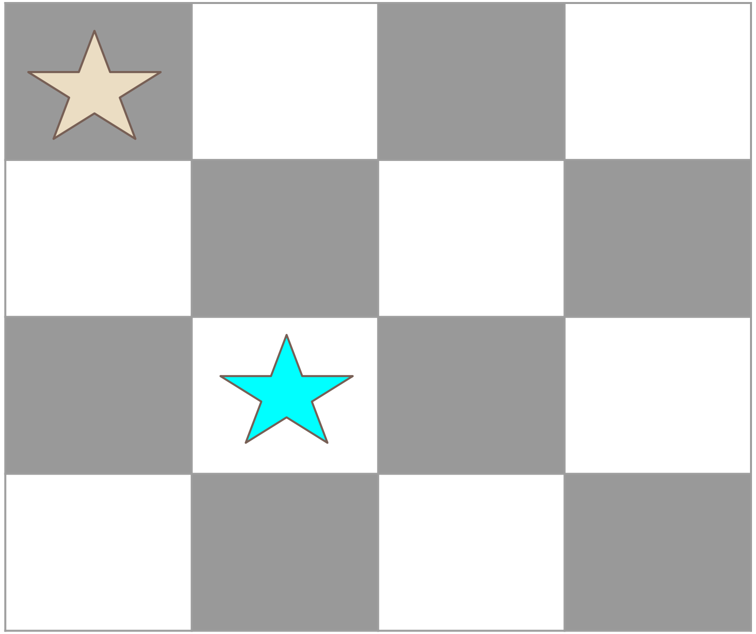
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);


      }
      }
   Return success;
```

Try Queen 3 (0,0)

Frame solveQueen(3, chess)



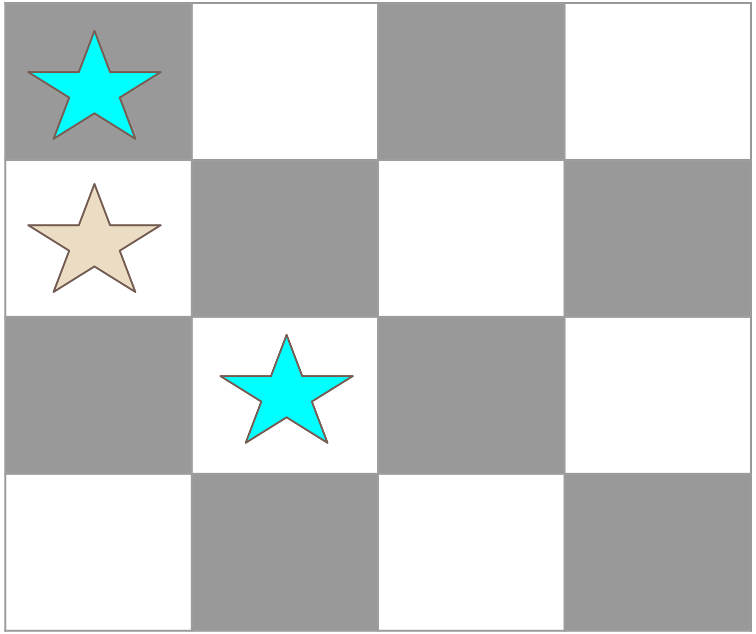
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);


      }
      }
   Return success;
```

Try Queen 3 (0,1)

Frame solveQueen(2, chess)



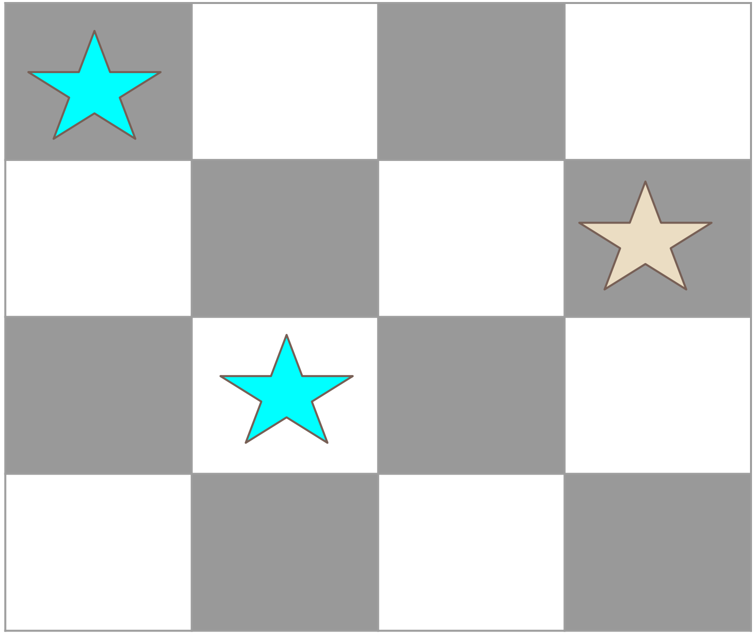
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);


      }
      }
   Return success;
```

Try Queen 3 until (1,3)

Frame solveQueen(2, chess)



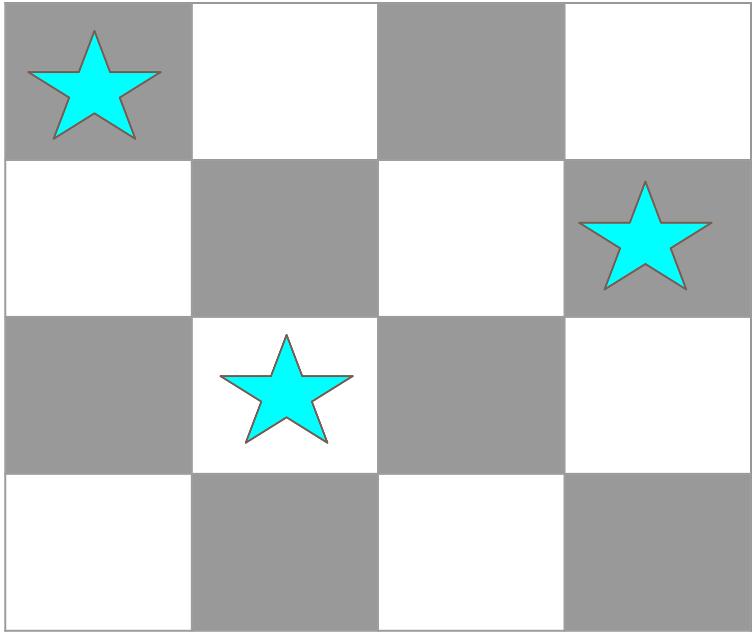
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(1,currentArray);


         }
         }
   Return success;
```

Is she safe? **Yes!**

Frame solveQueen(2, chess)



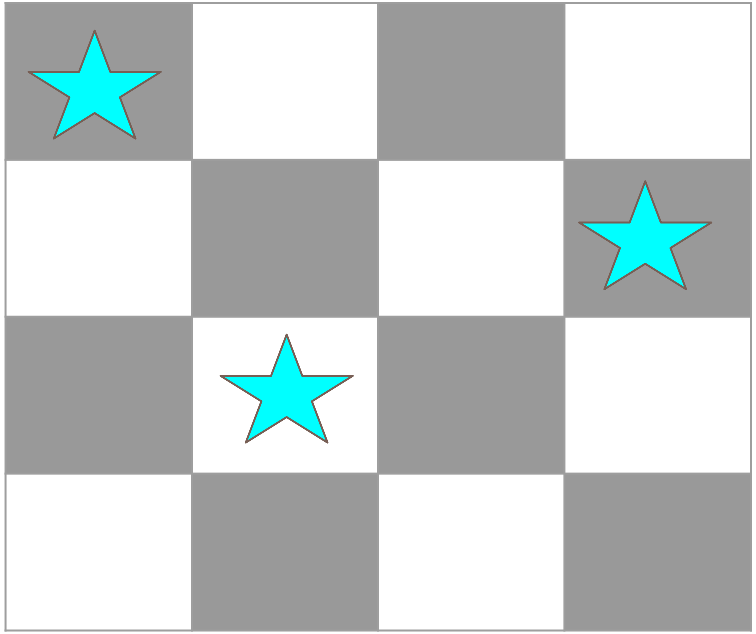
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);


      }
      }
   Return success;
```

Update chessboard

Frame solveQueen(2, chess)



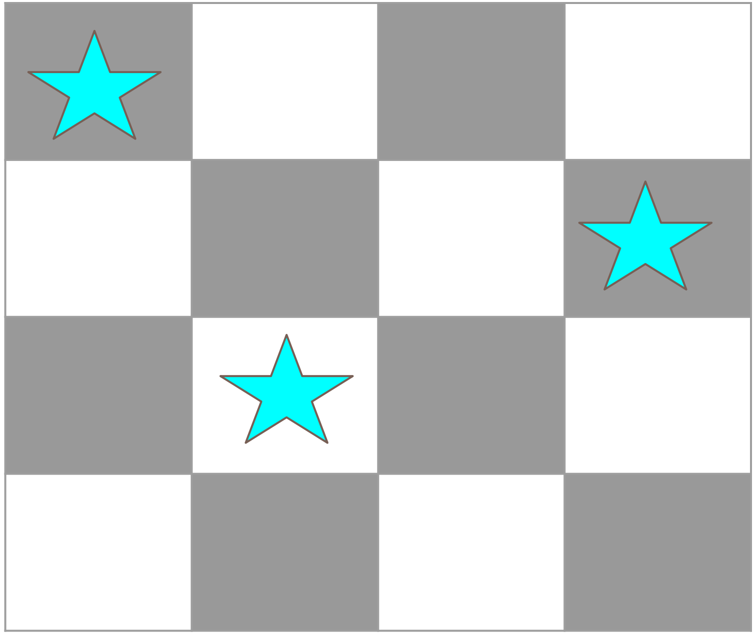
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(1,currentArray);


         }
         }
    Return success;
```

Call queen(1,currentArray);

Frame solveQueen(3, chess)



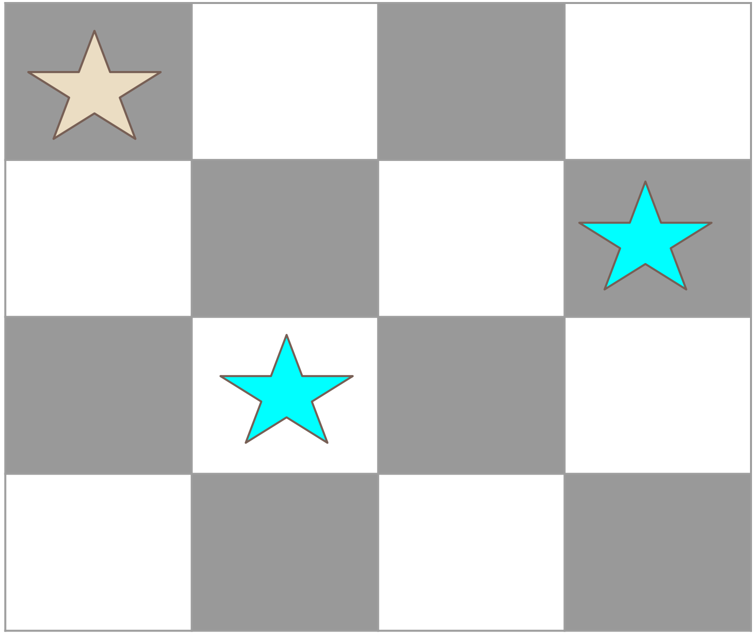
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
        bool success = isSafe(currentArray[row][col])
        If (success) {
           chess[row][col] = 1;
           success  = queen(0,currentArray);


        }
        }
   Return success;
```

Try Queen 4 (0,0)

Frame solveQueen(1, chess)

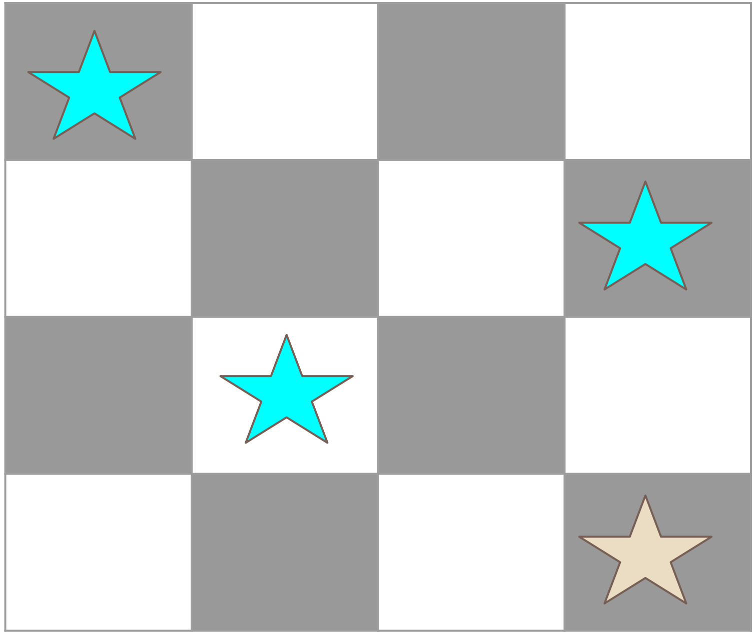


8-queens problem

Frame solveQueen(1, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(0,currentArray);


      }
      }
   Return success;
```



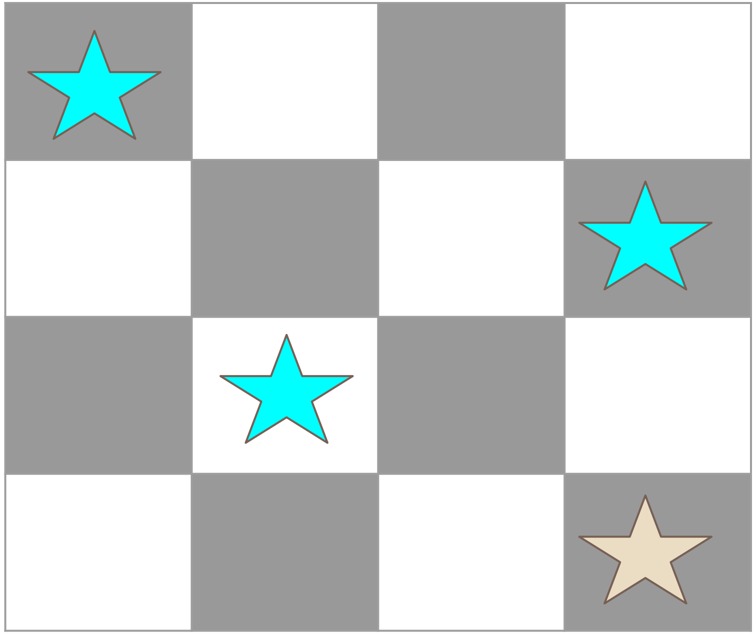
Try Queen 4 up to (3,3)

8-queens problem

Frame solveQueen(1, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(0,currentArray);


         }
         }
    Return success;
```



Queen cannot be placed!

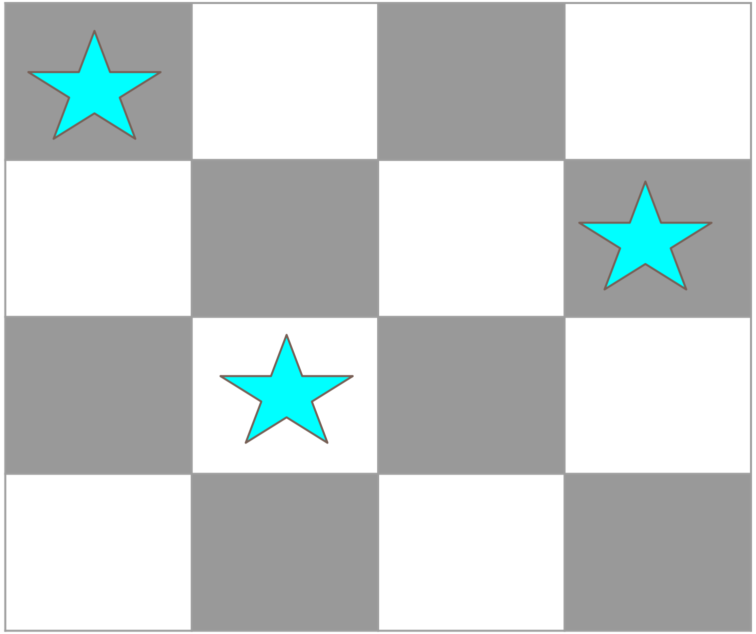
8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(0,currentArray);


         }
         }
    Return success;
```

Must backtrack. Return with success = false

Frame solveQueen(1, chess)

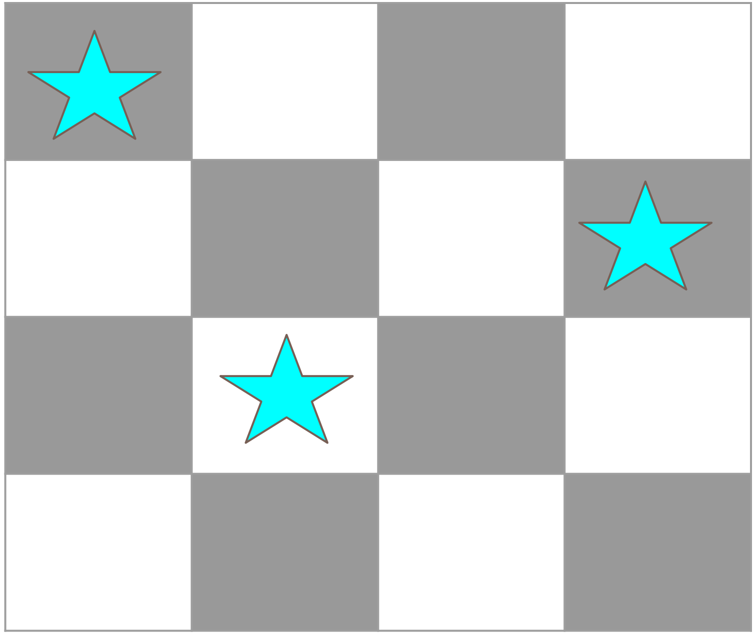


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Remove Queen 3. Continue trying to place it

Frame solveQueen(2, chess)

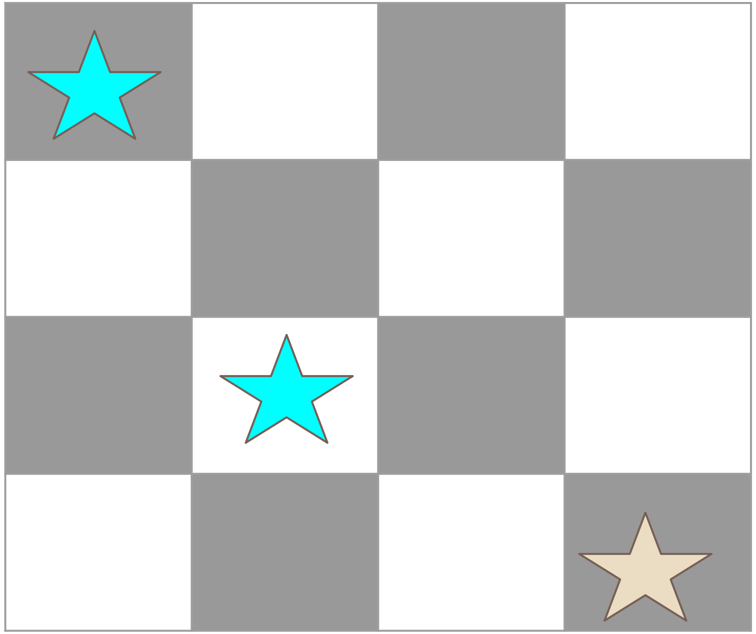


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Try Queen 3 (3,3)

Frame solveQueen(2, chess)

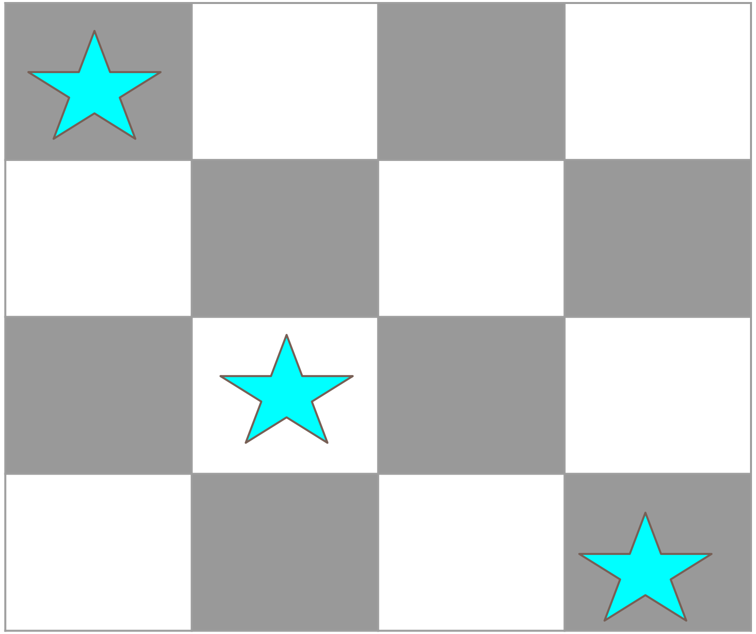


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Is Queen Safe? Yes

Frame solveQueen(2, chess)

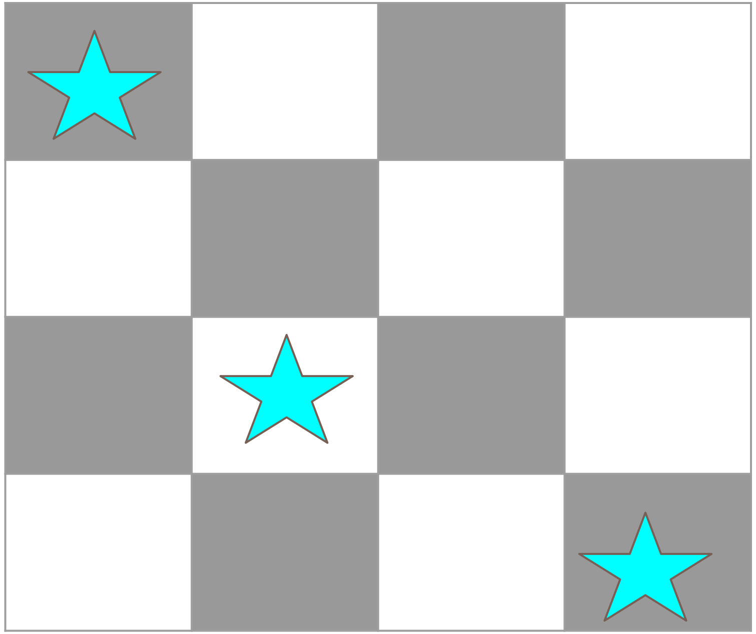


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Update chessboard

Frame solveQueen(2, chess)

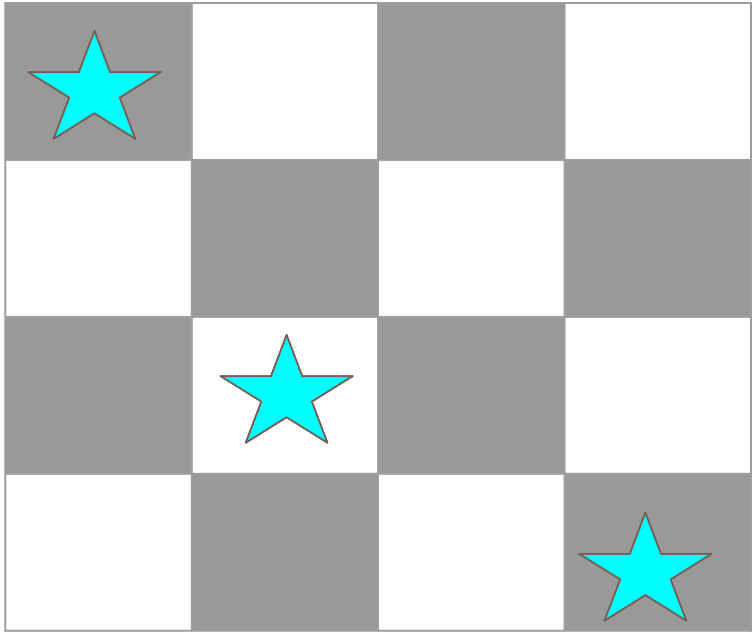


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Call queen(1,currentArray);

Frame solveQueen(2, chess)

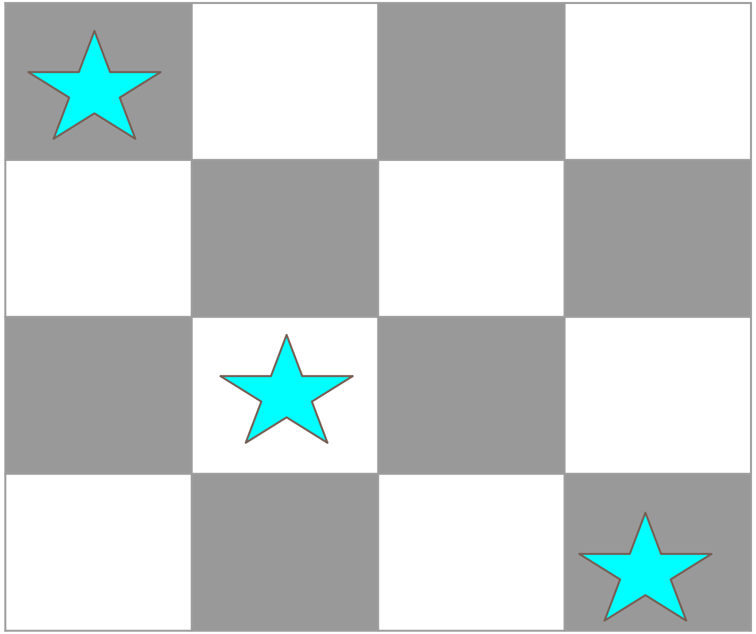


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(0,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Try all possibilities

Frame solveQueen(1, chess)

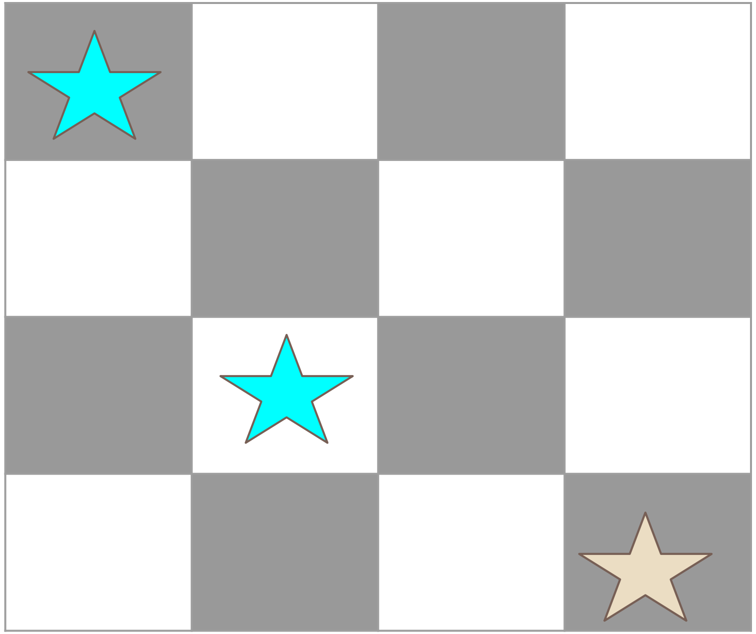


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(0,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Queen is never safe

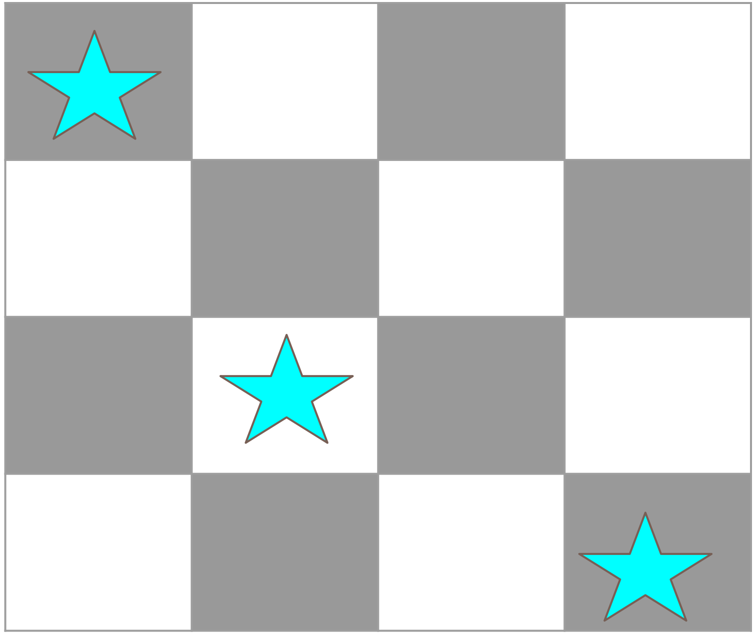
Frame solveQueen(1, chess)



8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(0,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
 Return success;
```

Frame solveQueen(1, chess)

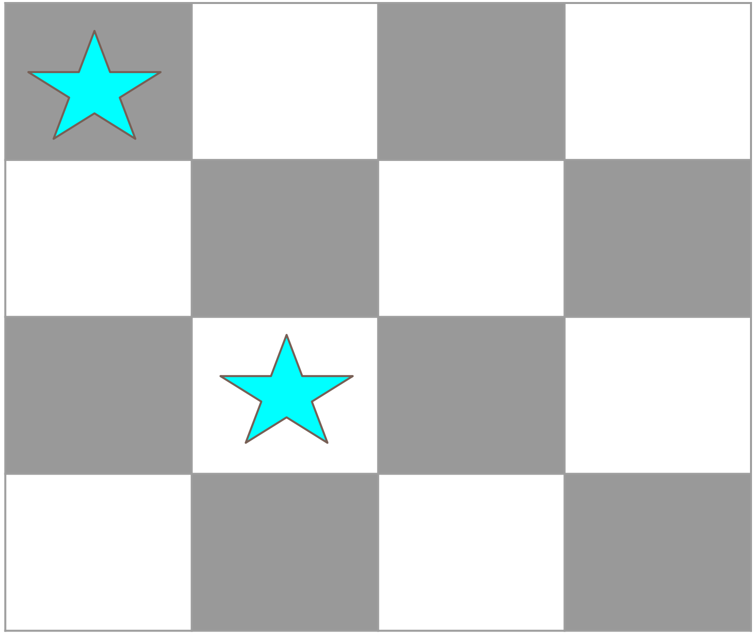


Backtrack with return success = false

8-queens problem

Frame solveQueen(2, chess)

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```



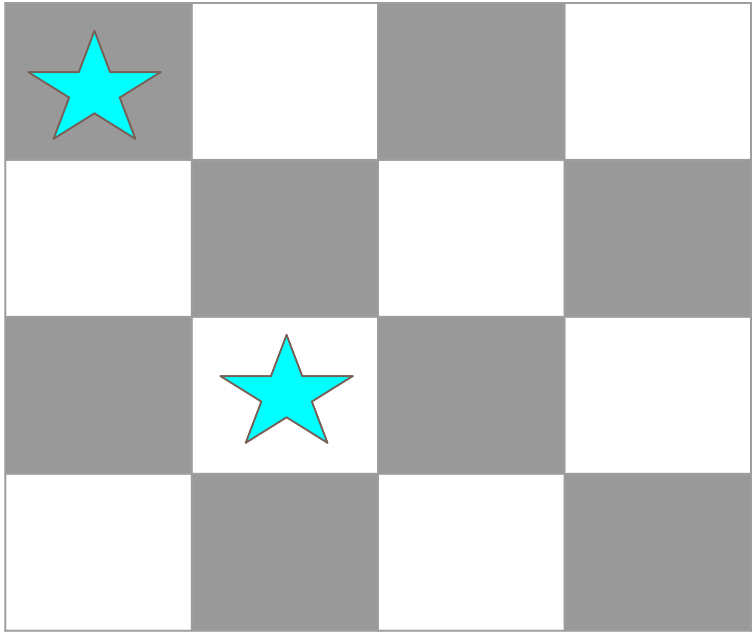
Undo third queen

8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

No more options for third queen.
Backtrack success = false

Frame solveQueen(2, chess)

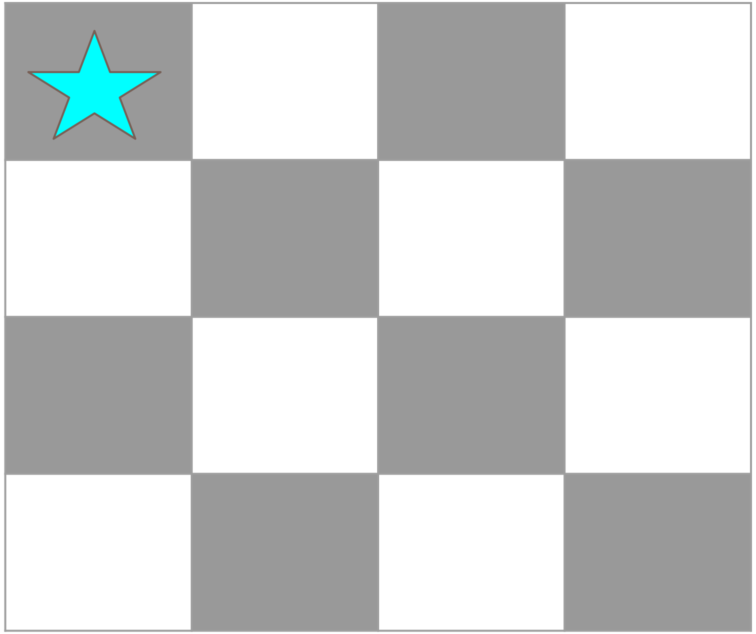


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(2,currentArray);
            If (!success) currentArray[row][col] = 0
         }
         }
    Return success;
```

Undo Second Queen

Frame solveQueen(3, chess)

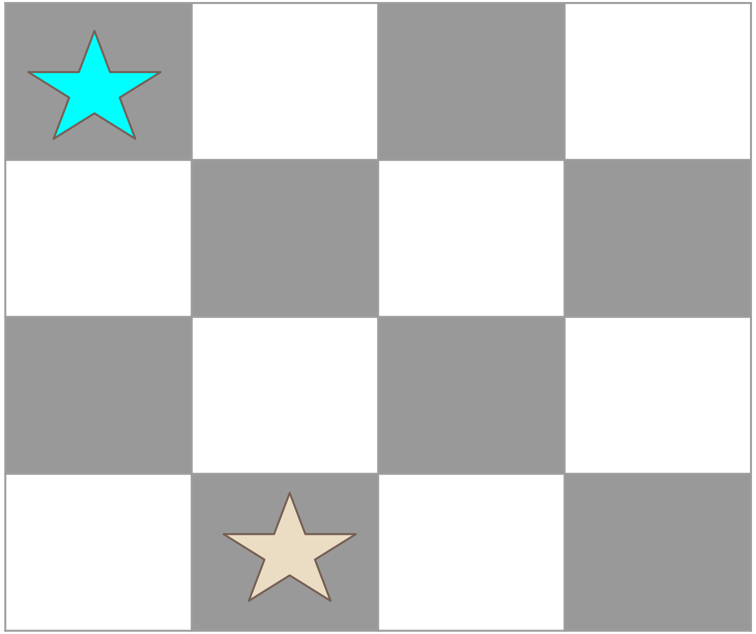


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Try Queen 2 (1,3)

Frame solveQueen(3, chess)

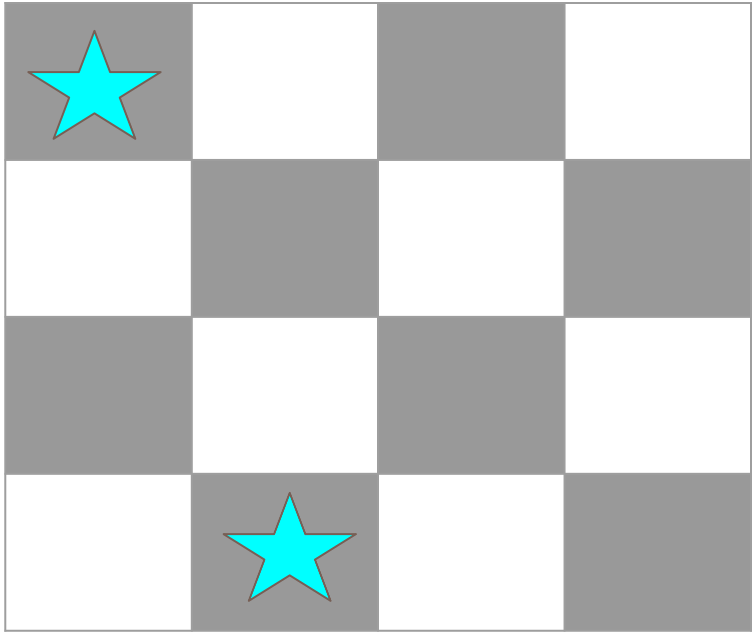


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(2,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Is Safe? Yes

Frame solveQueen(3, chess)

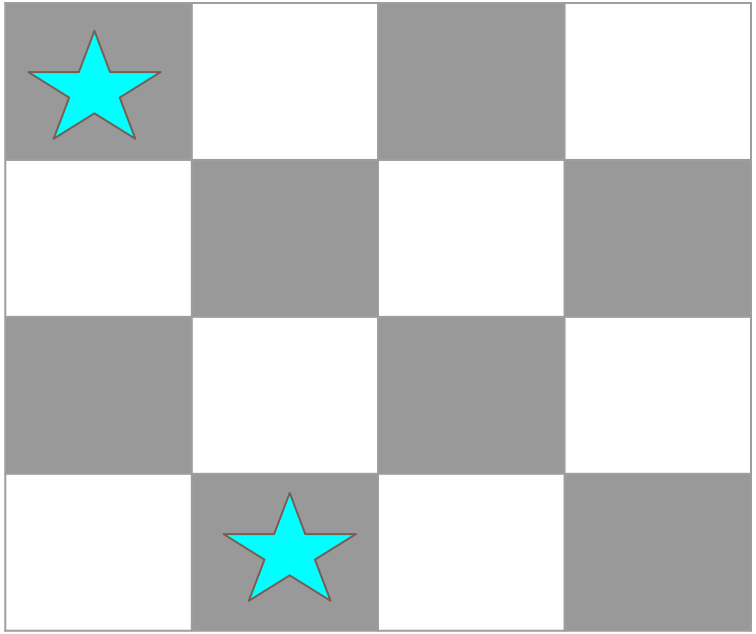


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(2,currentArray);
            If (!success) currentArray[row][col] = 0
         }
         }
   Return success;
```

Update Chessboard

Frame solveQueen(3, chess)

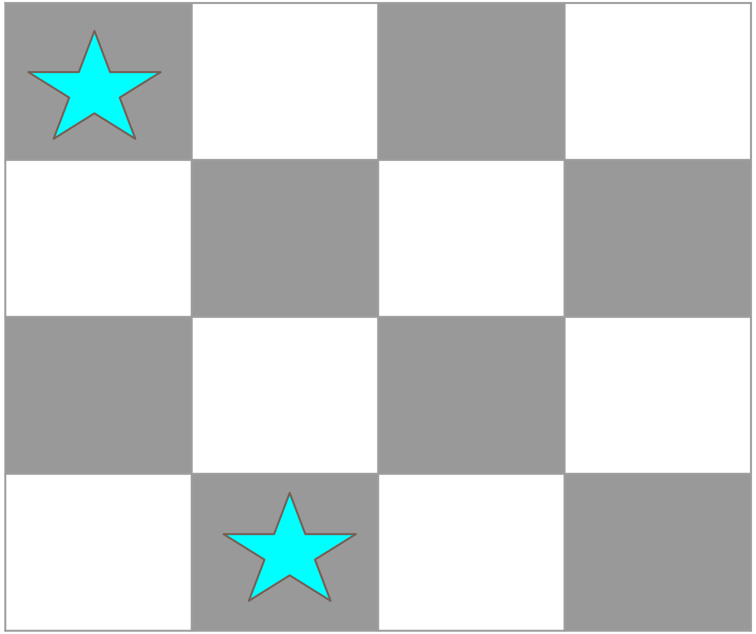


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(2,currentArray);
            If (!success) currentArray[row][col] = 0
         }
         }
  Return success;
```

Call queen(2,currentArray)

Frame solveQueen(3, chess)

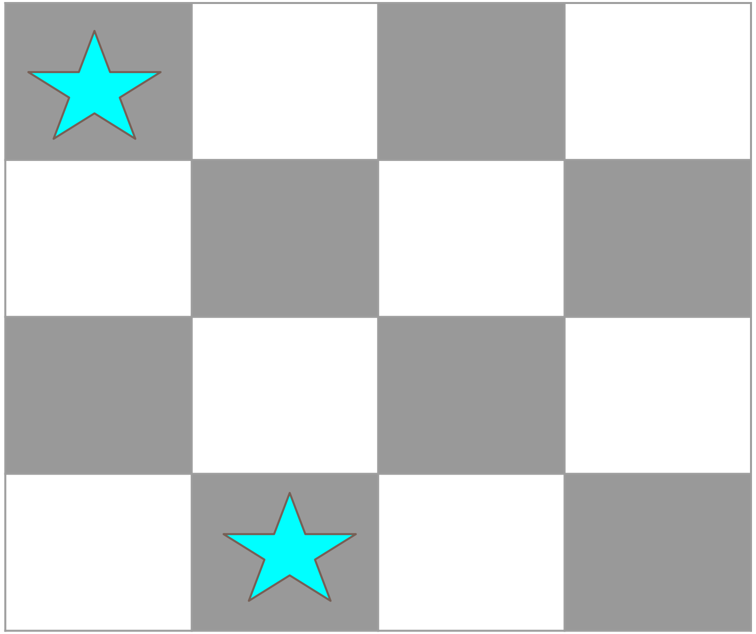


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
    Return success;
```

Try to place Queen 3

Frame solveQueen(2, chess)

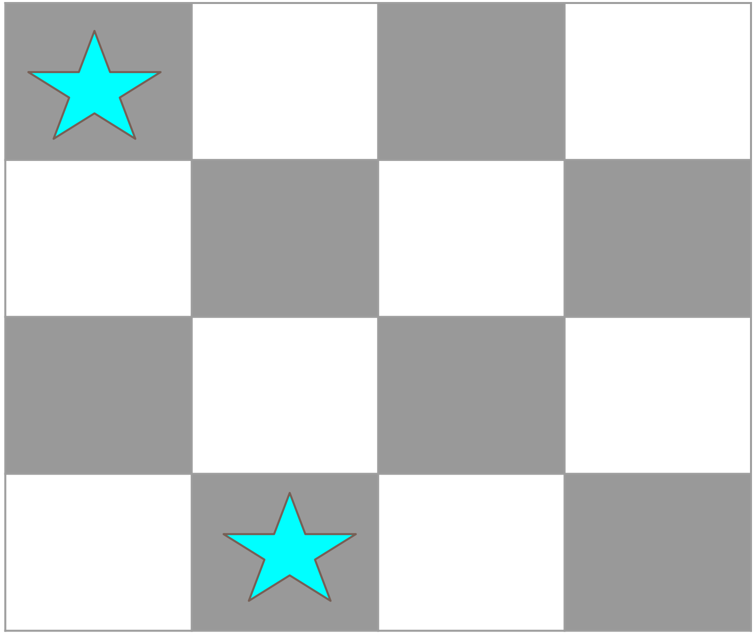


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

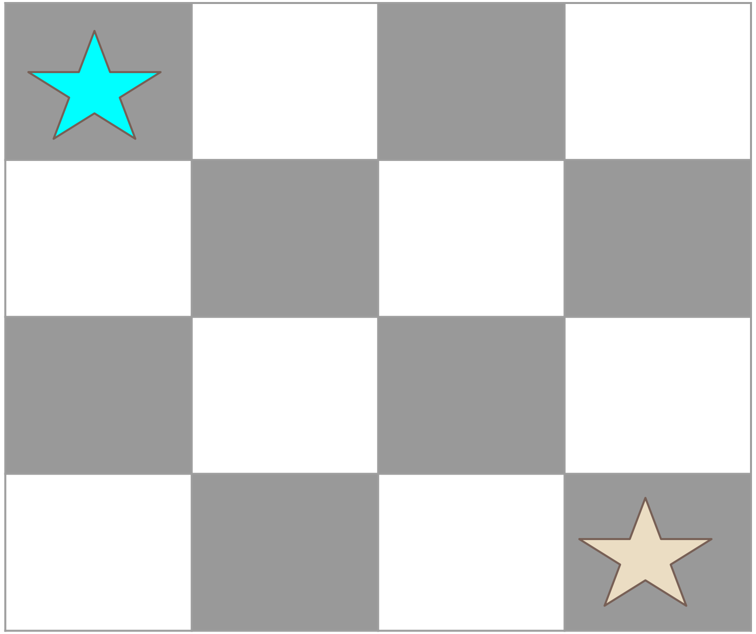
Try to place Queen 3

Frame solveQueen(2, chess)



8-queens problem

Fast forward a little bit: explore all solutions until reach last position of Queen 2. That solution also fails, so backtrack all the way back to Queen 1

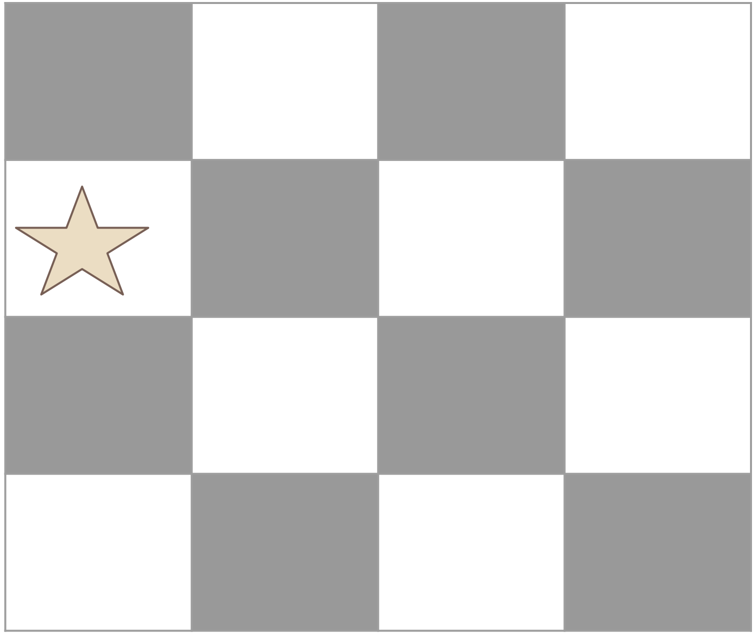


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(3,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Try to place Queen 1 (1,0)

Frame solveQueen(4, chess)

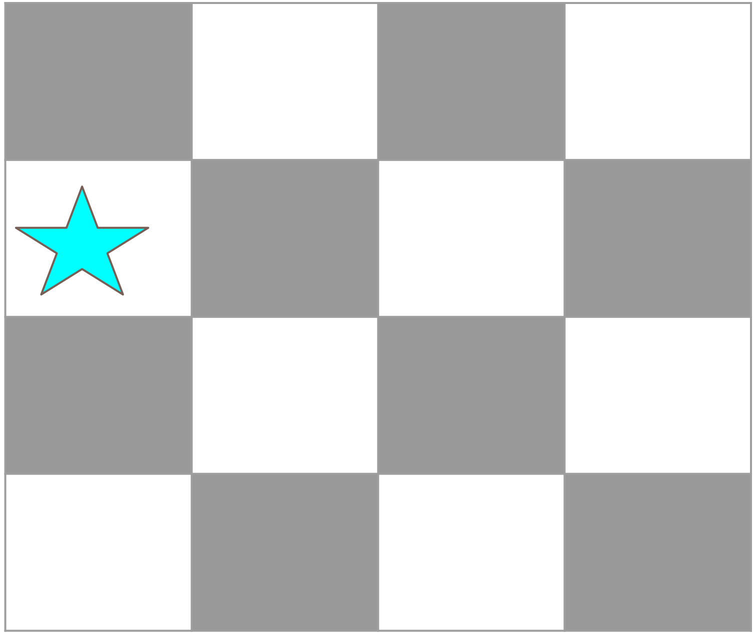


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(3,currentArray);
            If (!success) currentArray[row][col] = 0
         }
         }
    Return success;
```

Is it Safe? Yes

Frame solveQueen(4, chess)

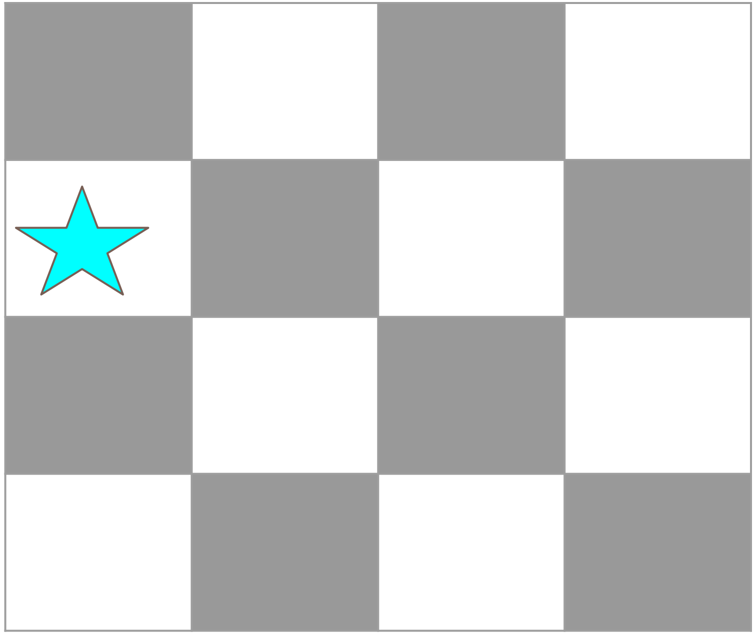


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(3,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Update chessboard

Frame solveQueen(4, chess)

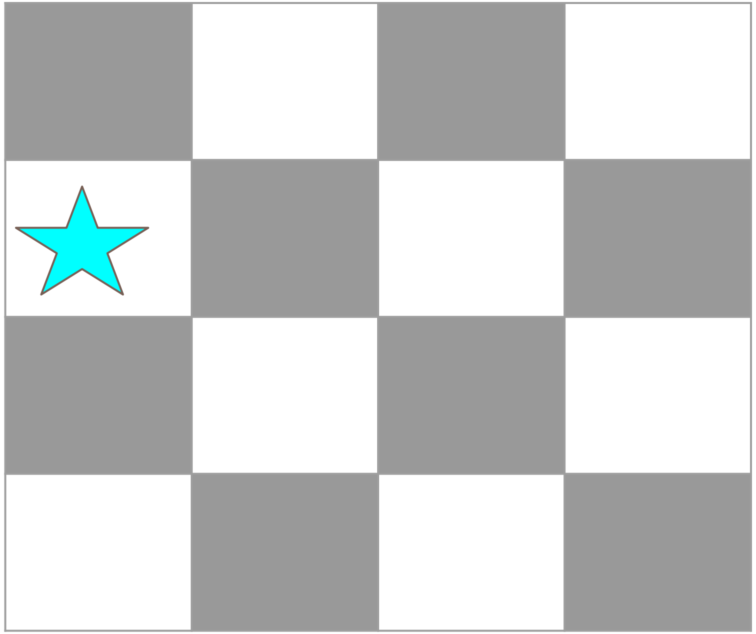


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(3,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Call queen(3,currentArray)

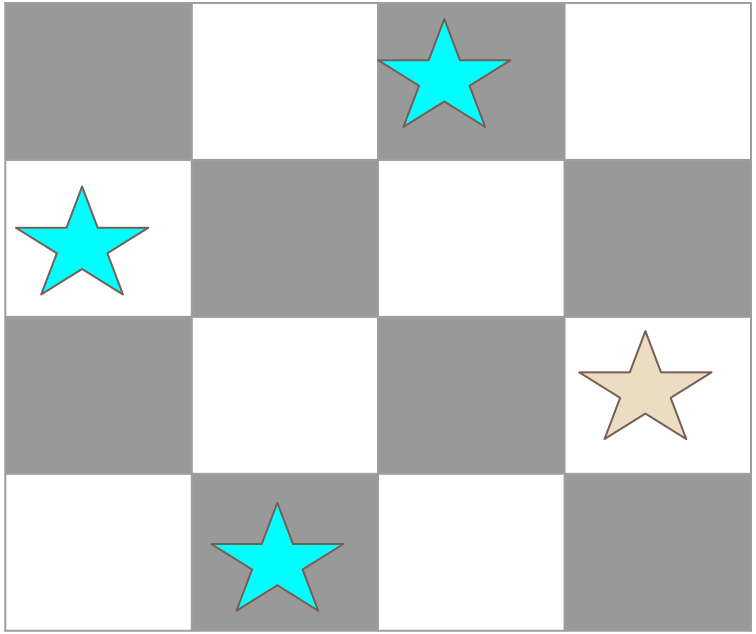
Frame solveQueen(4, chess)



8-queens problem

Frame solveQueen(1, chess)

Fast forward a little, go through every solution, backtracking as necessary, until are in the following state



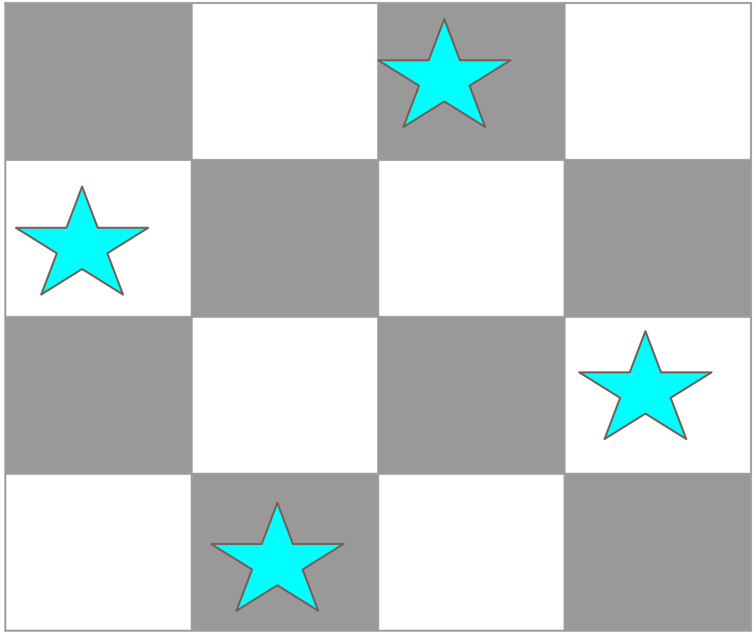
Place Queen 4

8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(0,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

It is safe!

Frame solveQueen(1, chess)

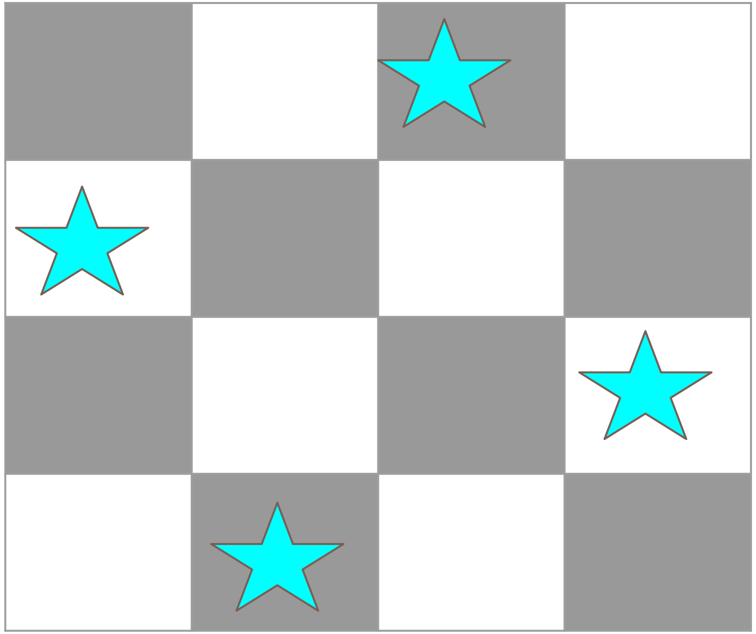


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(0,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Update chessboard

Frame solveQueen(1, chess)

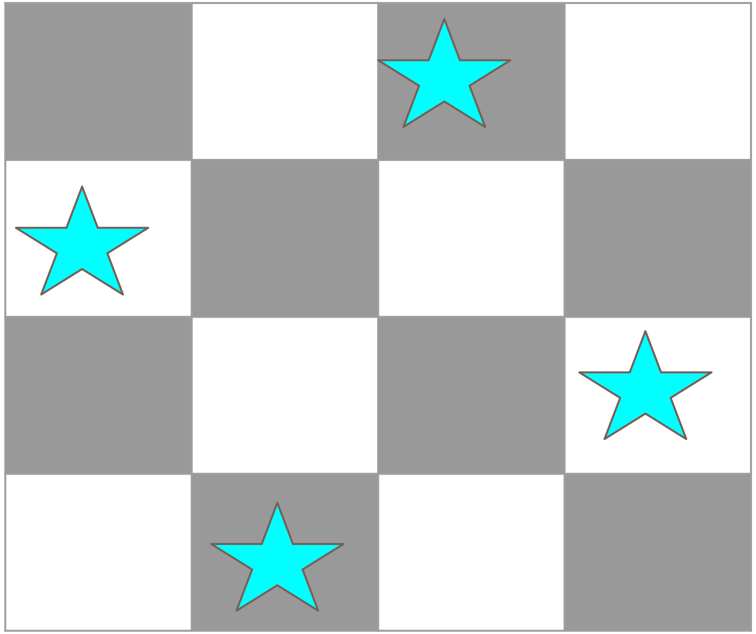


8-queens problem

```
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(0,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
   Return success;
```

Call queen(0,currentArray)

Frame solveQueen(1, chess)

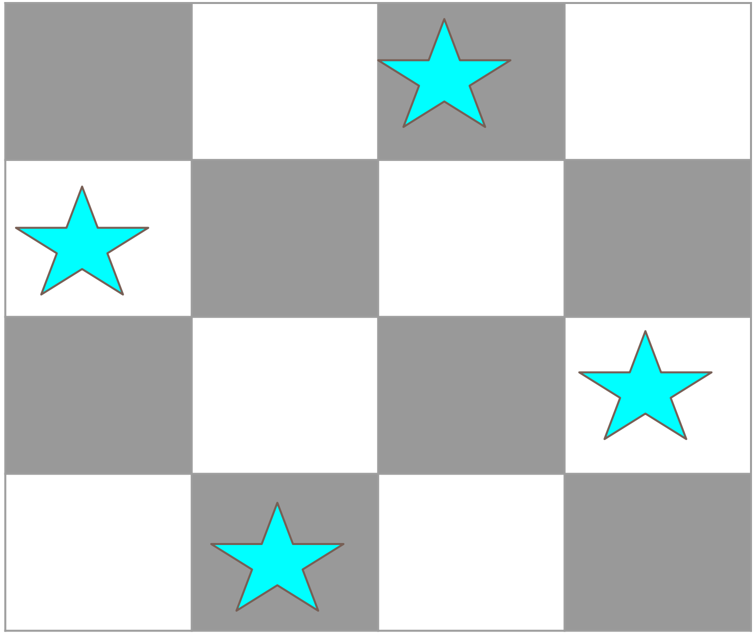


8-queens problem

```
if (n ==0) return true
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(-1,currentArray);
        If (!success) currentArray[row][col] = 0
      }
      }
  Return success;
```

Hit base case

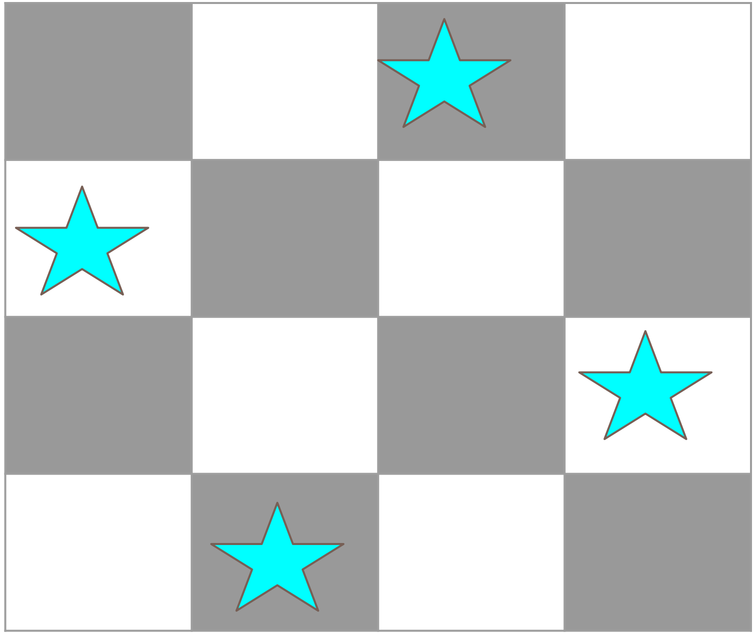
Frame solveQueen(0, chess)



8-queens problem

```
if (n ==0) return true
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
        bool success = isSafe(currentArray[row][col])
        If (success) {
          chess[row][col] = 1;
          success  = queen(0,currentArray);
          If (!success) currentArray[row][col] = 0
          else return true;
        }
        }
   Return success;
```

Frame solveQueen(1, chess)



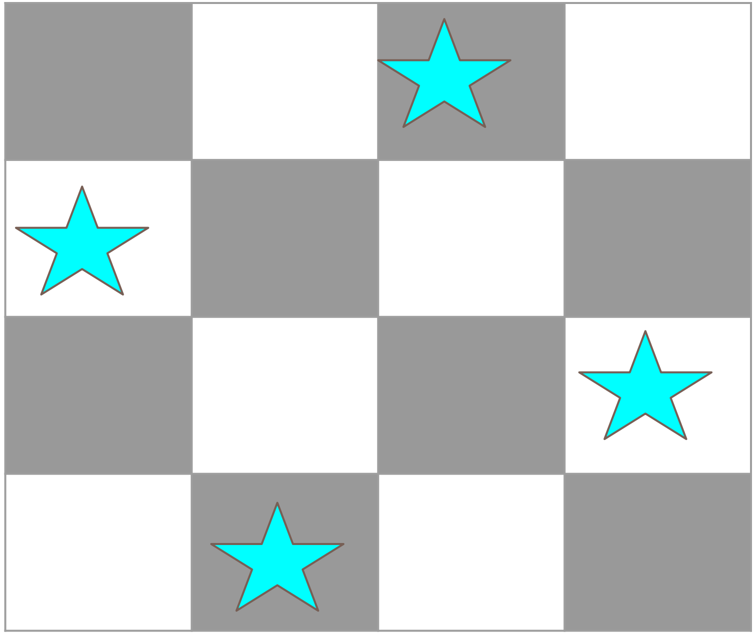
Return true in previous frame

8-queens problem

```
if (n ==0) return true
for (int row = 0 ; row < chessboard.length ; row++) {
  for (int col = 0 ; col < chessboard.length ; col++) {
      bool success = isSafe(currentArray[row][col])
      If (success) {
        chess[row][col] = 1;
        success  = queen(1,currentArray);
        If (!success) currentArray[row][col] = 0
        else return true;
      }
      }
   Return success;
```

Return true in previous frame

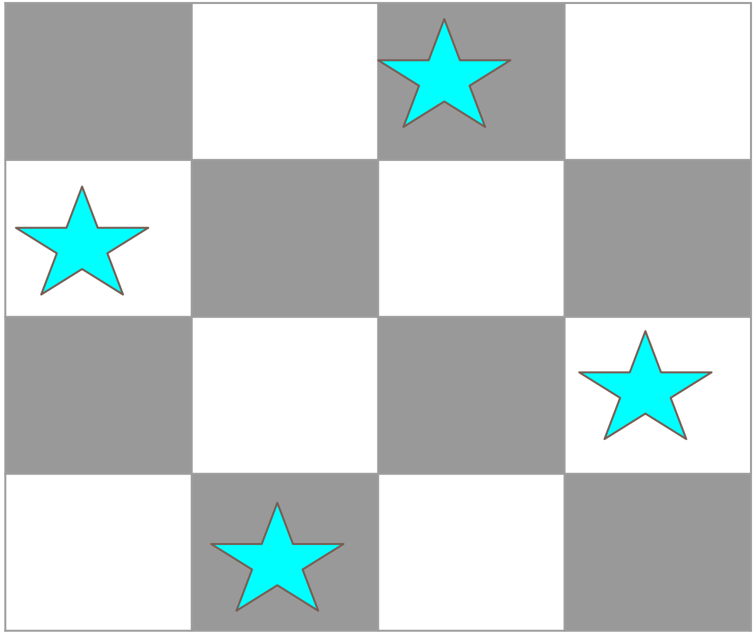
Frame solveQueen(2, chess)



8-queens problem

```
if (n ==0) return true
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(2,currentArray);
            If (!success) currentArray[row][col] = 0
            else return true;
         }
         }
    Return success;
```

Frame solveQueen(3, chess)

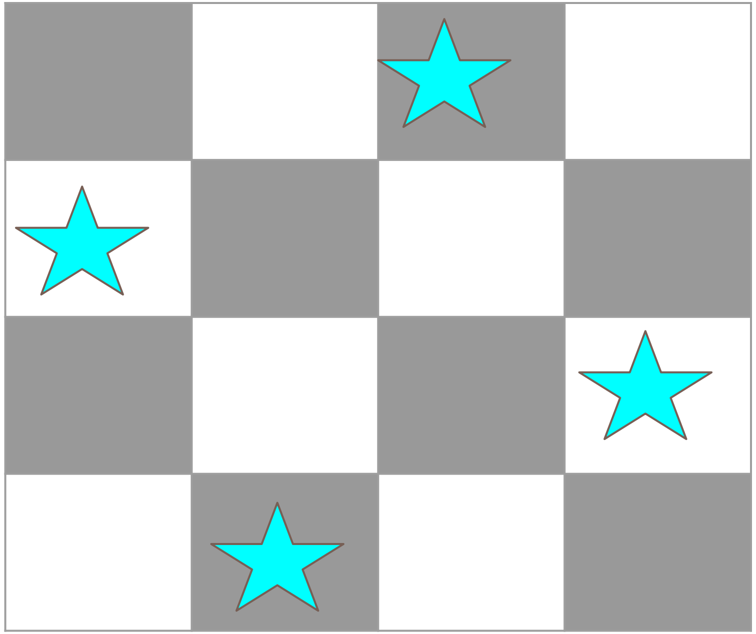


Return true in previous frame

8-queens problem

```
if (n ==0) return true
for (int row = 0 ; row < chessboard.length ; row++) {
   for (int col = 0 ; col < chessboard.length ; col++) {
         bool success = isSafe(currentArray[row][col])
         If (success) {
            chess[row][col] = 1;
            success  = queen(3,currentArray);
            If (!success) currentArray[row][col] = 0
            else return true;
         }
         }
    Return success;
```

Frame solveQueen(4, chess)

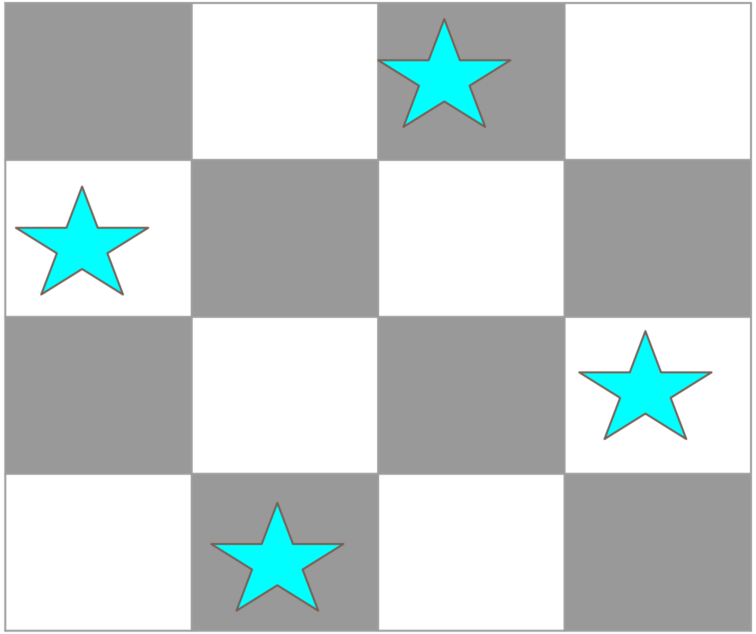


Return true in previous frame

8-queens problem

Frame solveQueen(4, chess)

```
Chess =
{ {0,0,1,0},
   {1,0,0,0},
   {0,0,0,1},
   {0,1,0,0}
}
```



Return true in function!

8-queens problem

```
1) Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column. For every row do:
    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 umark this [row, column] (Backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger backtracking.
```

8-queens problem

- Using the algorithm described, try implementing it in Java for HW

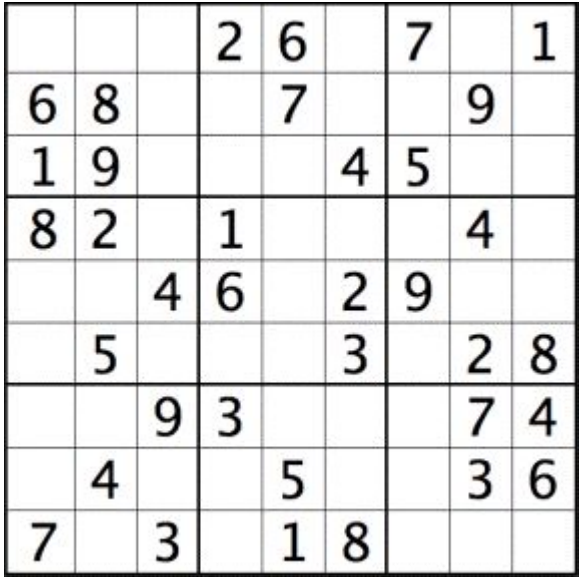
- Hint 1: First define a method: isSafe(int[][] board) that determines whether a queen can be placed here. You need to check that there is no queen in the current row, column, diagonal

- Hint 2: if placing a queen in a particular location fails (because other queens can't be placed there in the recursive calls), remember to remove the queen from that location

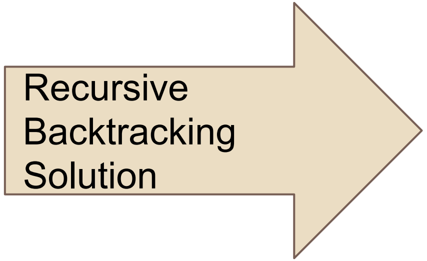
```
public static boolean solve(int nbQueens, int[][] board)
```

```
// empty entries marked as zero. Entries with queens marked as 1
```

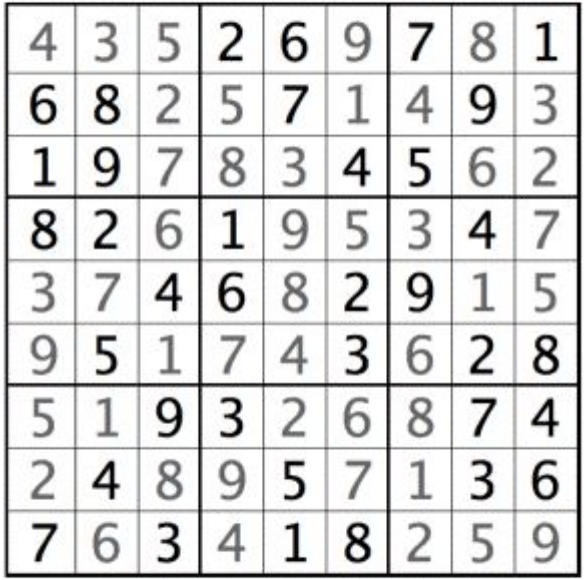
Sudoku (Solve it for a bonus in HW!)



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



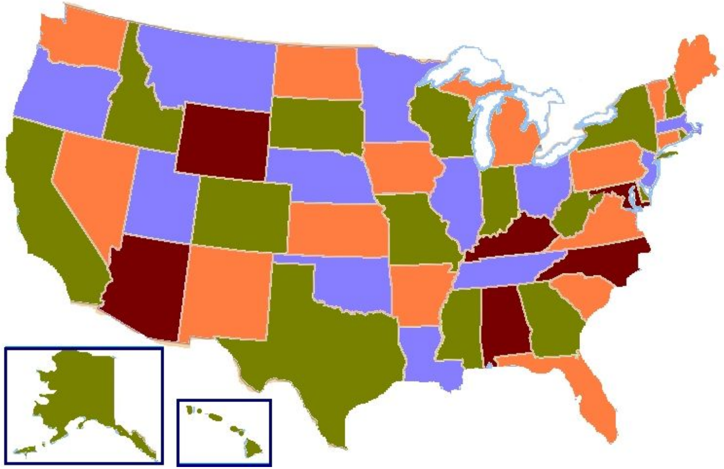
Recursive
Backtracking
Solution



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

```
public static boolean solve(int[][] s) // empty entries marked as zero.
```

Map Colouring Problem

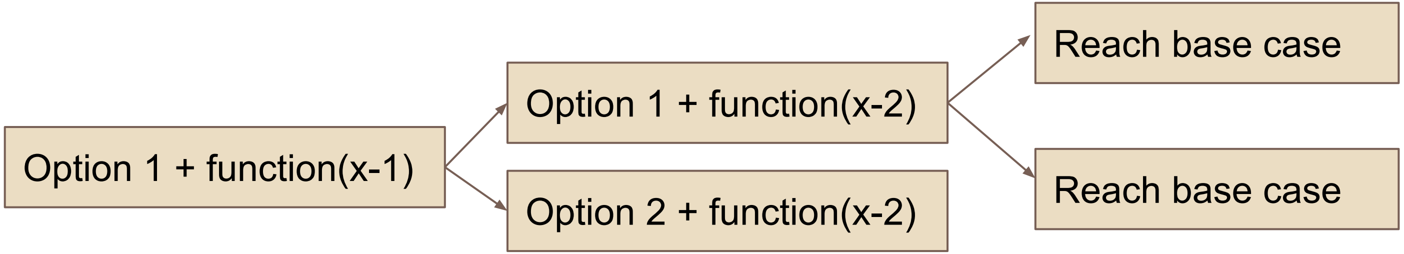


Cool theorem: given any separation of a plane into contiguous regions, producing a figure called a *map*, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color

But how do you find the assignment?
You can use recursive backtracking, just like we did for the other solutions

Tons of cool applications: AI, register assignment in computer architecture, etc.

Remember:
Recursion with backtracking (Type 2)



```
function(x) =
    If base case { if (isValid(x)) return true ; else false}
    else { // Recursive Step
        for every option:
            if (function(x-1) == true) {
                // Solution found
                return true
        }
```

Stops as soon as has found a solution => it is called greedy

Will recursion save the world?

- Recursion makes it easy to solve complex algorithms and explore problems with many different parameters/constraints
  - Easy to solve 8-Queens, Sudoku, etc.

- **But:** it can be a very expensive solution!
  - Think about a sudoku solution: for each square, I try 9 solutions. Then another 9, then another 9 …. 9 x 9 x 9 … gets expensive pretty quickly!

- Next lecture, we will **formalise** what **expensive** actually means!