Section 2

CSCI E-22

Will Begin Shortly

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = ____________________;

    ...



}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = ____________________;

    ...



}
```

```
removeCapitals("HeLLo")
```

```
return "eo"
```

Recall: Recursive Problem Solving

When solving problems using recursion, we break the problem down into smaller subproblems.

Once we have broken down the problem into the smallest subproblem (one that we can solve), we have reached a base case.

Then, we can progressively build up the solutions to the subproblems until we have a solution for the overall problem.

```
sum(n) =
```

Recall: Recursive Problem Solving

When solving problems using recursion, we break the problem down into smaller subproblems.

Once we have broken down the problem into the smallest subproblem (one that we can solve), we have reached a base case.

Then, we can progressively build up the solutions to the subproblems until we have a solution for the overall problem.

```
sum(n) = n + sum(n-1)
```

Recall: Recursive Problem Solving

When solving problems using recursion, we break the problem down into smaller subproblems.

Once we have broken down the problem into the smallest subproblem (one that we can solve), we have reached a base case.

Then, we can progressively build up the solutions to the subproblems until we have a solution for the overall problem.

```
sum(n) = n + sum(n-1)
```

```
public static int sum(int n) {
    if (n <= 0)
        return 0;

    int rest = sum(n - 1);
    return n + rest;
}
```

Recall: Recursive Problem Solving

When solving problems using recursion, we break the problem down into smaller subproblems.

Once we have broken down the problem into the smallest subproblem (one that we can solve), we have reached a base case.

Then, we can progressively build up the solutions to the subproblems until we have a solution for the overall problem.

```
sum(n) = n + sum(n-1)
```

```
public static int sum(int n) {
    if (n <= 0)
        return 0;

    int rest = sum(n - 1);
    return n + rest;
}
```

```
sum(4) = 4 + sum(3)
  sum(3) = 3 + sum(2)
    sum(2) = 2 + sum(1)
      sum(1) = 1 + sum(0)
        sum(0) = 0
```

Recall: Recursive Problem Solving

When solving problems using recursion, we break the problem down into smaller subproblems.

Once we have broken down the problem into the smallest subproblem (one that we can solve), we have reached a base case.

Then, we can progressively build up the solutions to the subproblems until we have a solution for the overall problem.

```
sum(n) = n + sum(n-1)
```

```
public static int sum(int n) {
    if (n <= 0)
        return 0;

    int rest = sum(n - 1);
    return n + rest;
}
```

```
sum(4) = 4 + sum(3)
  sum(3) = 3 + sum(2)
    sum(2) = 2 + sum(1)
      sum(1) = 1 + sum(0)
        sum(0) = 0
```

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
- What's the recursive subproblem?
- What work do we need to do before returning?

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = ____________________;

    ...



}
```

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
- What's the recursive subproblem?
- What work do we need to do before returning?

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
  - When `s` is the empty string
- What's the recursive subproblem?
- What work do we need to do before returning?

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
  - When `s` is the empty string
- What's the recursive subproblem?
  - Removing the capitals from the rest of the string
- What work do we need to do before returning?

```
removeCapitals("HeLLo")
```

If we knew the solution to `removeCapitals("eLLo")`, we could solve `removeCapitals("HeLLo")`

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
  - When `s` is the empty string
- What's the recursive subproblem?
  - Removing the capitals from the rest of the string
- What work do we need to do before returning?

```
removeCapitals("HeLLo") = H + removeCapitals("eLLo")
                          ?
```

If we knew the solution to `removeCapitals("eLLo")`, we could solve `removeCapitals("HeLLo")`

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
  - When `s` is the empty string
- What's the recursive subproblem?
  - Removing the capitals from the rest of the string
- What work do we need to do before returning?

`removeCapitals("HeLLo") = H` ? `+ removeCapitals("eLLo")` `"eo"`

If we knew the solution to `removeCapitals("eLLo")`, we could solve `removeCapitals("HeLLo")`

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
  - When `s` is the empty string
- What's the recursive subproblem?
  - Removing the capitals from the rest of the string
- What work do we need to do before returning?
  - Test whether the first character in the string is a capital; if it is, it should not be included in the return value

`removeCapitals("HeLLo") = H` ? `+ removeCapitals("eLLo")` `"eo"`

If we knew the solution to `removeCapitals("eLLo")`, we could solve `removeCapitals("HeLLo")`

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
  - When `s` is the empty string
- What's the recursive subproblem?
  - Removing the capitals from the rest of the string
- What work do we need to do before returning?
  - Test whether the first character in the string is a capital; if it is, it should not be included in the return value

```
removeCapitals("HeLLo") = H + removeCapitals("eLLo")
                                "eo"
```

If we knew the solution to `removeCapitals("eLLo")`, we could solve `removeCapitals("HeLLo")`

Recursive Problem Solving Approach

Before writing a recursive method, we can try to plan:

- What's the base case?
  - When `s` is the empty string
- What's the recursive subproblem?
  - Removing the capitals from the rest of the string
- What work do we need to do before returning?
  - Test whether the first character in the string is a capital; if it is, it should not be included in the return value


```
removeCapitals("HeLLo") = H + removeCapitals("eLLo") = "eo"
                                "eo"
```


If we knew the solution to `removeCapitals("eLLo")`, we could solve `removeCapitals("HeLLo")`

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = ____________________;

    ...



}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    ...



}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (____________________)



}
```


Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:


```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return _________________;


}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return _________________;
}
```


Practice with Recursion

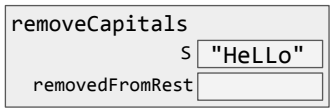
The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:


```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

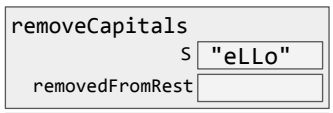
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

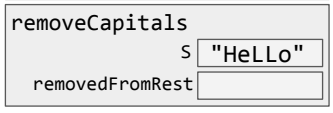


Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

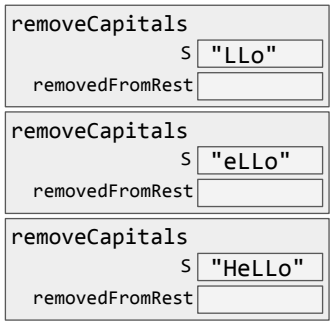




Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

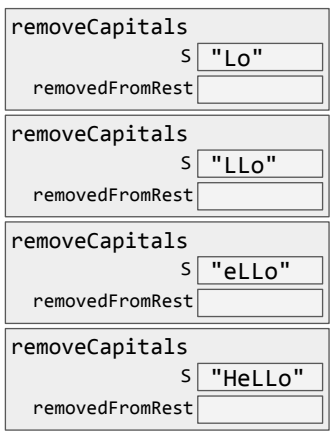
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```



Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

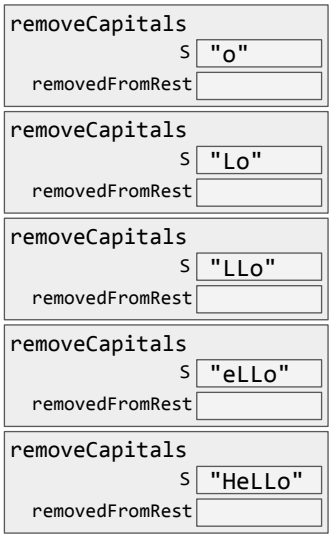
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```



Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

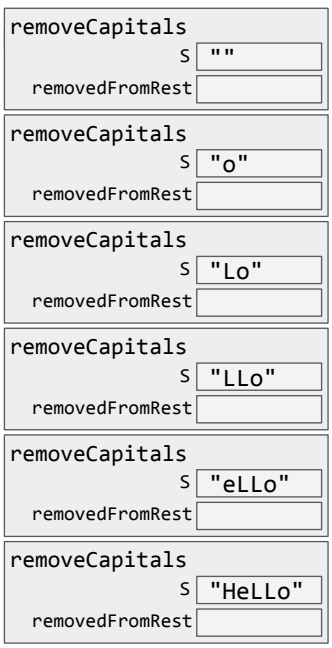
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```



Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

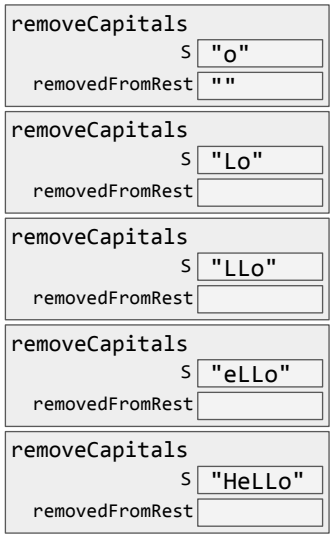
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

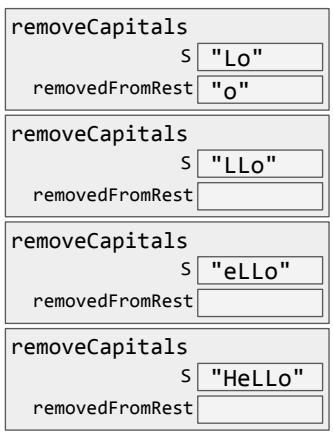
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```



Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

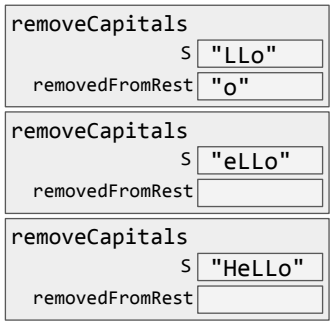
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

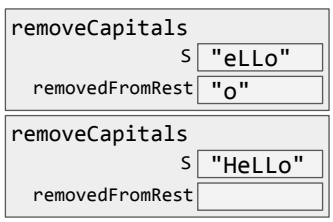
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```



Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

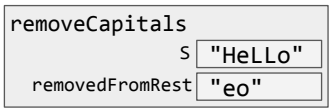
```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```



Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0?

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0? 1

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0? 1

n is 1?

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0? 1
n is 1? 2

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0? 1
n is 1? 2
n is 2? 3
n is 3? 4

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0? 1
n is 1? 2
n is 2? 3
n is 3? 4

What is the general formula for the number of calls needed to process a string of length n?

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0? 1
n is 1? 2
n is 2? 3
n is 3? 4

What is the general formula for the number of calls needed to process a string of length n?

n + 1

Practice with Recursion

The following Java method should use recursion to remove all capital letters from a string. Some of the code has been omitted and we will need to fill it in:

```
public static String removeCapitals(String s) {
    if (s == null)
        throw new IllegalArgumentException();

    if (s.equals(""))
        return "";

    String removedFromRest = removeCapitals(s.substring(1));

    char first = s.charAt(0);
    if (first >= 'A' && first <= 'Z')
        return removedFromRest;
    else
        return first + removedFromRest;
}
```

How many calls of this method are needed when…

n is 0? 1
n is 1? 2
n is 2? 3
n is 3? 4

What is the general formula for the number of calls needed to process a string of length n?

n + 1

Let's remember this while we complete the next exercise

The Fibonacci Sequence

The Fibonacci sequence is a well-known number series in which each number in the series is the sum of the two previous numbers.

We define the first two numbers as $F_0 = 0$ and $F_1 = 1$, and all successive numbers as:

$$F_n = F_{n-1} + F_{n-2}$$

Since the sequence is defined recursively, using recursion to calculate Fibonacci numbers is natural. Finish the code below to write an algorithm that calculates the *n*th Fibonacci number.

```
public static long fib(int n) {

    // base case
    if (_________________)




 }
```

The Fibonacci Sequence

The Fibonacci sequence is a well-known number series in which each number in the series is the sum of the two previous numbers.

We define the first two numbers as $F_0 = 0$ and $F_1 = 1$, and all successive numbers as:

$$F_n = F_{n-1} + F_{n-2}$$

Since the sequence is defined recursively, using recursion to calculate Fibonacci numbers is natural. Finish the code below to write an algorithm that calculates the *n*th Fibonacci number.

```
public static long fib(int n) {

    // base case
    if (n == 0 || n == 1)
        return _________;



}
```


The Fibonacci Sequence

The Fibonacci sequence is a well-known number series in which each number in the series is the sum of the two previous numbers.

We define the first two numbers as $F_0 = 0$ and $F_1 = 1$, and all successive numbers as:

$$F_n = F_{n-1} + F_{n-2}$$

Since the sequence is defined recursively, using recursion to calculate Fibonacci numbers is natural. Finish the code below to write an algorithm that calculates the *n*th Fibonacci number.


```
public static long fib(int n) {

    // base case
    if (n == 0 || n == 1)
        return n;



}
```

The Fibonacci Sequence

The Fibonacci sequence is a well-known number series in which each number in the series is the sum of the two previous numbers.

We define the first two numbers as $F_0 = 0$ and $F_1 = 1$, and all successive numbers as:

$$F_n = F_{n-1} + F_{n-2}$$

Since the sequence is defined recursively, using recursion to calculate Fibonacci numbers is natural. Finish the code below to write an algorithm that calculates the *n*th Fibonacci number.

```
public static long fib(int n) {

    // base case
    if (n == 0 || n == 1)
        return n;

    // recursive case
    return ____________________;

}
```


The Fibonacci Sequence

The Fibonacci sequence is a well-known number series in which each number in the series is the sum of the two previous numbers.

We define the first two numbers as $F_0 = 0$ and $F_1 = 1$, and all successive numbers as:

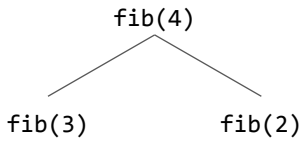
$$F_n = F_{n-1} + F_{n-2}$$

Since the sequence is defined recursively, using recursion to calculate Fibonacci numbers is natural. Finish the code below to write an algorithm that calculates the *n*th Fibonacci number.


```
public static long fib(int n) {

    // base case
    if (n == 0 || n == 1)
        return n;

    // recursive case
    return fib(n - 1) + fib(n - 2);

}
```

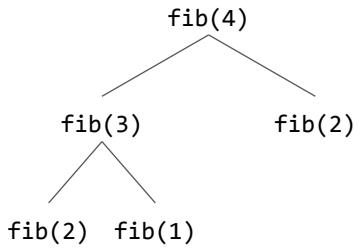
The Fibonacci Sequence

Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:



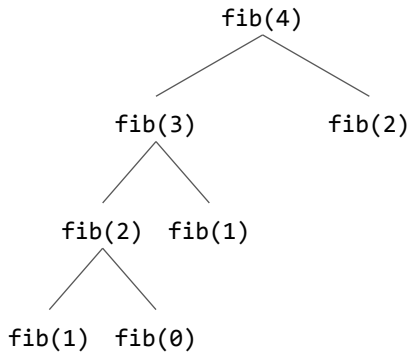
The Fibonacci Sequence

Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:



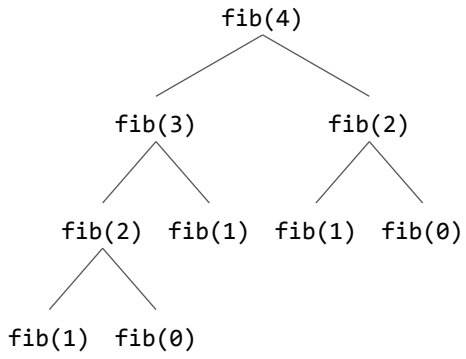
The Fibonacci Sequence

Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:



The Fibonacci Sequence

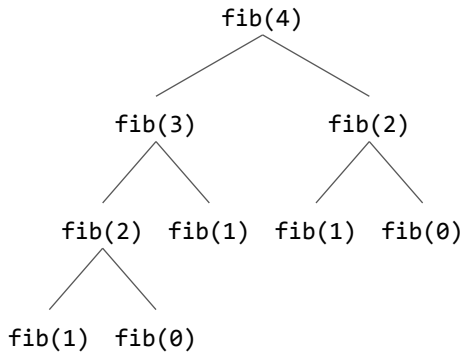
Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:



The Fibonacci Sequence

Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:

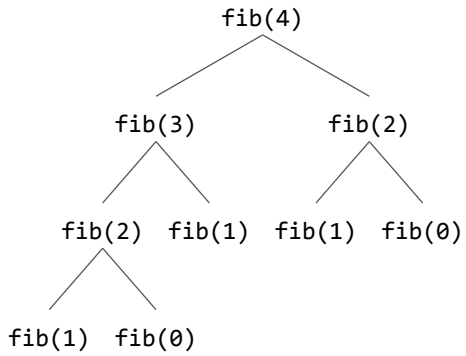
How many times did we call `fib()` to find the fourth Fibonacci number?



The Fibonacci Sequence

Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:

How many times did we call `fib()` to find the fourth Fibonacci number? 9

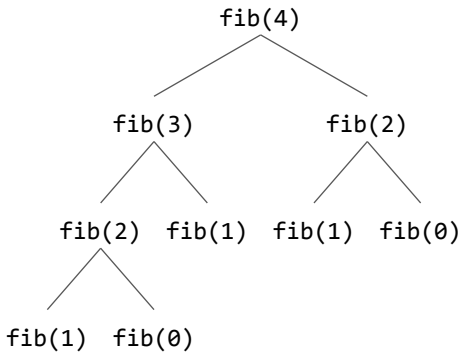


The Fibonacci Sequence

Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:

How many times did we call `fib()` to find the fourth Fibonacci number? 9

Do you see a problem with this? What if we tried larger numbers, like 50?



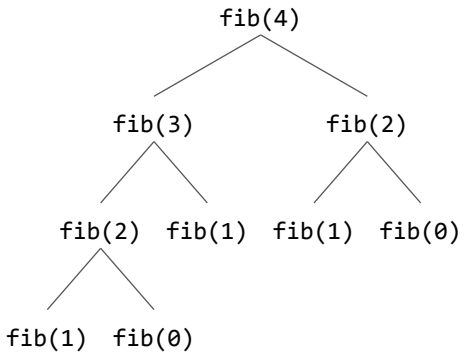
The Fibonacci Sequence

Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:

How many times did we call `fib()` to find the fourth Fibonacci number? 9

Do you see a problem with this? What if we tried larger numbers, like 50?

- The number of calls increases exponentially for the initial value of *n*. This means that we would have to make about 10 billion method calls to calculate `fib(50)`!



The Fibonacci Sequence

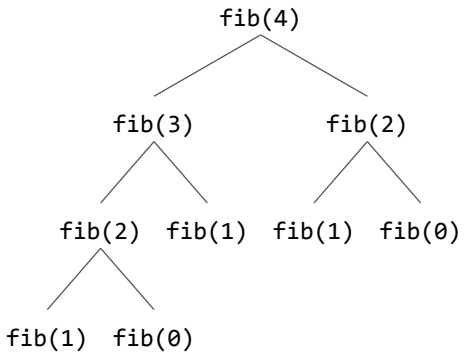
Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:

How many times did we call `fib()` to find the fourth Fibonacci number? 9

Do you see a problem with this? What if we tried larger numbers, like 50?

- The number of calls increases exponentially for the initial value of *n*. This means that we would have to make about 10 billion method calls to calculate `fib(50)`!

How would you rewrite `fib()` to be more efficient, either still as a recursive method or iteratively?



The Fibonacci Sequence

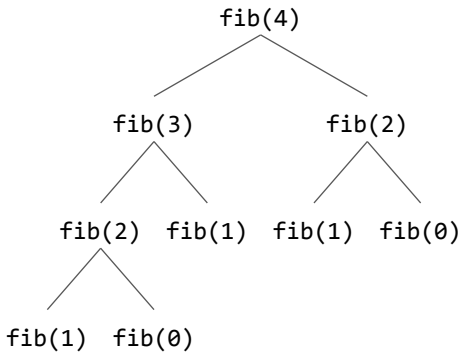
Draw a diagram that shows the number of times `fib()` is called with an initial value of 4. It has been started for you:

How many times did we call `fib()` to find the fourth Fibonacci number? 9

Do you see a problem with this? What if we tried larger numbers, like 50?

- The number of calls increases exponentially for the initial value of *n*. This means that we would have to make about 10 billion method calls to calculate `fib(50)`!

How would you rewrite `fib()` to be more efficient, either still as a recursive method or iteratively?



- You need to keep track of the previous two Fibonacci values while computing the next

The Fibonacci Sequence

Here is an example iterative solution:

```
public static long fib(int n) {

    if (n <= 0) {
        return 0;
    }


    long previous = 0;      // at the start, previous = F_0
    long current = 1;       // at the start, current = F_1


    for (int i = 2; i <= n; i++) {
        long tmp = previous + current;
        previous = current;
        current = tmp;
    }

    return current;
}
```

The Fibonacci Sequence

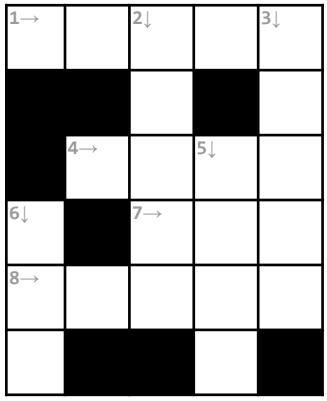
Here is an example recursive solution:

```
public static long fib(int n) {

    if (n == 0) {
        return 0;
    }

    return fibHelper(n, 1, 0);
}

public static long fibHelper(int n, long curr, long prev) {

    if (n == 1) {
        return curr;
    }

    return fibHelper(n - 1, curr + prev, curr);

}
```

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

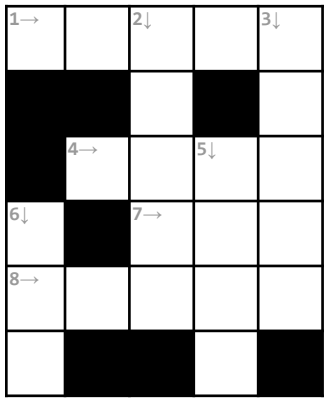
aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie

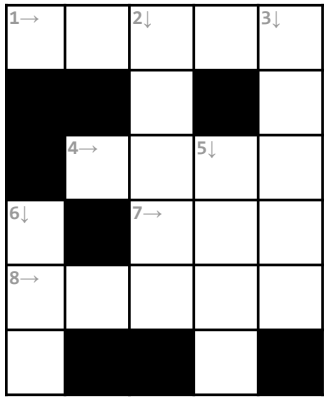



```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

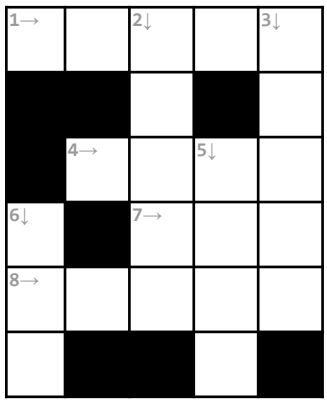
This call to `findSolution()` is trying to place a value in position n

In this problem, n identifies the slot we're trying to put a word in, e.g. 5 down

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

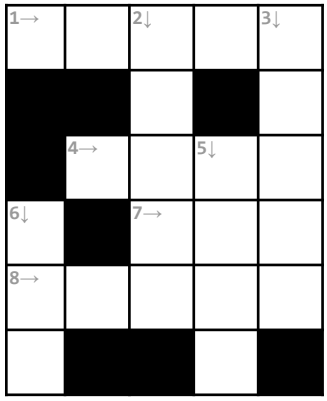
The base case is when a solution is found

In this problem, a solution is found when all 8 slots are filled

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

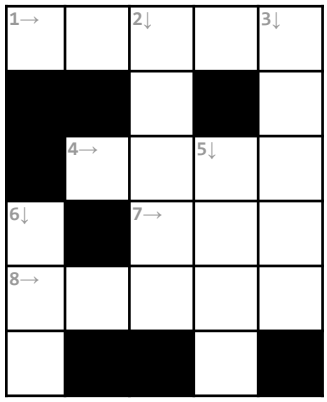
For all possible values that can be put in positions

In this problem, the words from the word list are the values

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

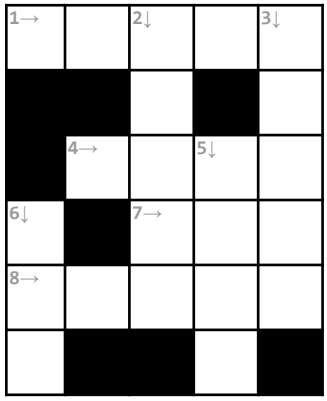
`isValid()` checks whether the current potential value can be placed in the position

In this problem, what are the constraints for whether a word can be placed in a slot?

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

`isValid()` checks whether the current potential value can be placed in the position

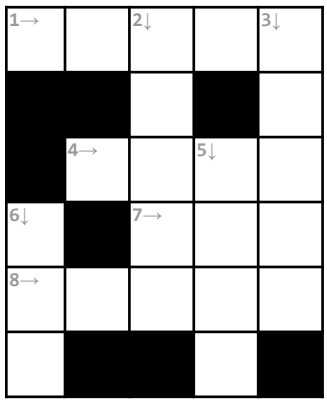
In this problem, what are the constraints for whether a word can be placed in a slot?

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



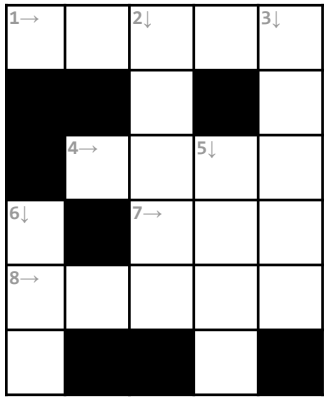

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

Add value to the board

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



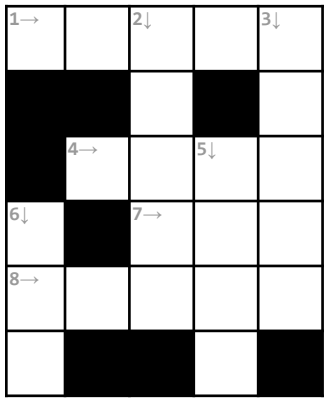

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

Make a recursive call to move on to next position (n+1) to try to continue to fill board

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

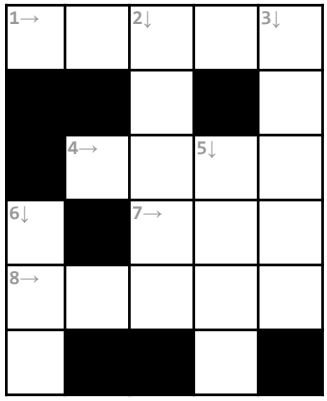
Make a recursive call to move on to next position (n+1) to try to continue to fill board

There are two possibilities of what happens next

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



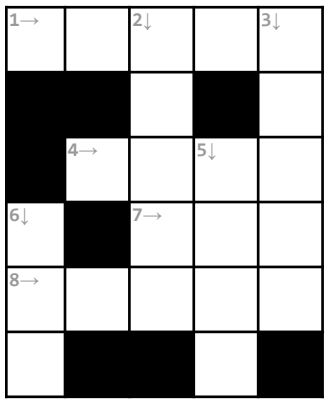

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

(1) If we continue to place values in positions until we find a solution, we return `true` in the base case and return `true` all the way back through the recursive call stack

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

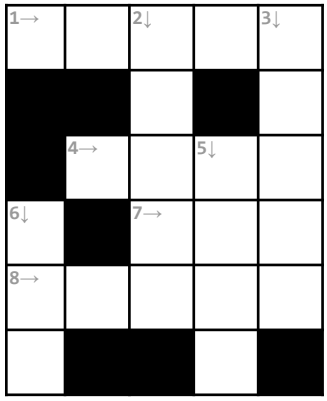
(2) If we reach a position `n` that we cannot fill -- because none of the values are valid for that position -- the `for` loop is exhausted

We then return `false`. We must backtrack!

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

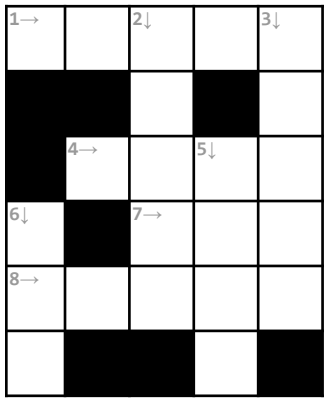
(2) When we backtrack, the recursive call returns false, indicating that placing that value on the board resulted in an unsolvable state

We must remove the value from the board and continue in the `for` loop to try to place other values

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie

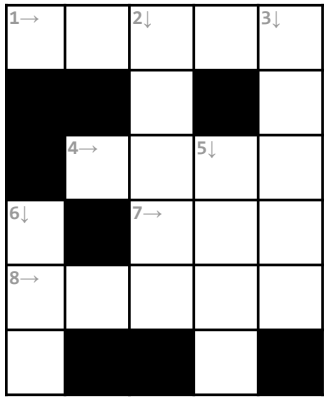



```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

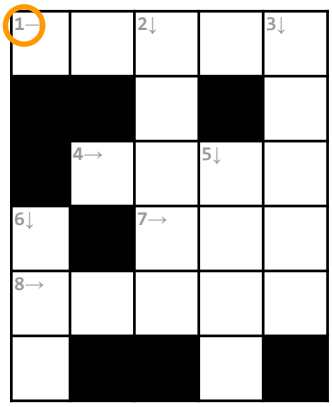
Notice that, in order to do a sample run, we need to assign an arbitrary order in which to assign the spaces in the crossword puzzle.

We use the following order for this run: 1ACROSS, 2DOWN, 3DOWN, 4ACROSS, 7ACROSS, 5DOWN, 8ACROSS, 6DOWN.

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



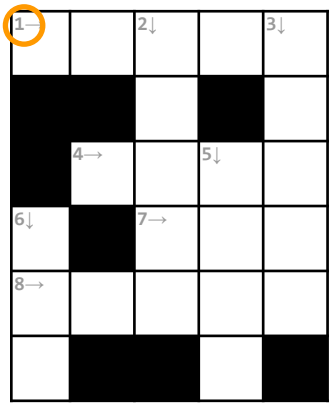

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```


1 across

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



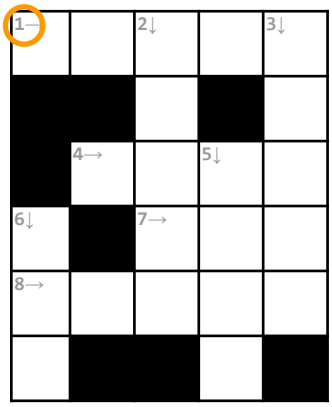

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

1 across

Recursive Backtracking

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

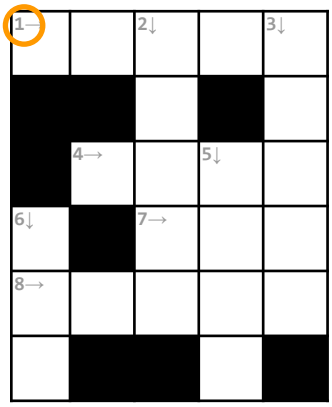
1 across

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

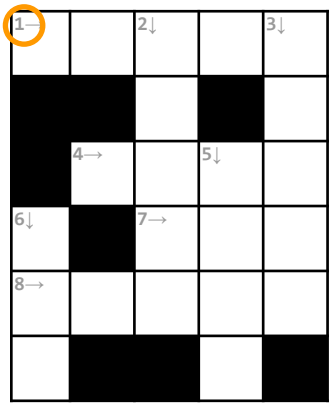
`1 across  val "aft"`

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

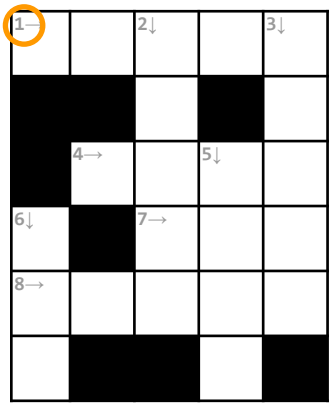
`1 across  val "ale"`

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

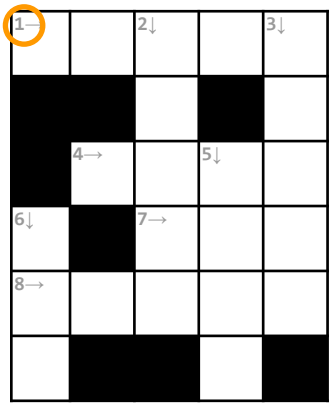
1 across val "eel"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

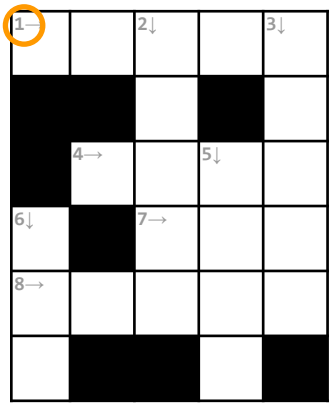
1 across val "heel"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

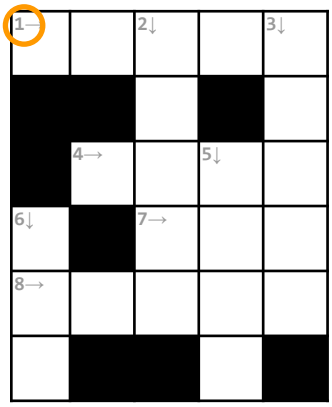
`1 across  val "hike"`

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

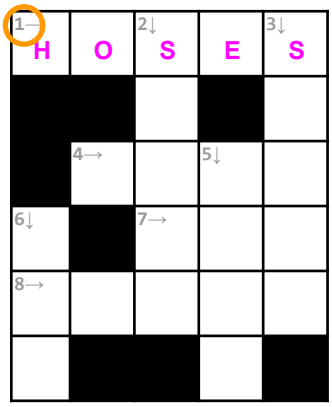

`1 across  val "hoses"`

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which ones are used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

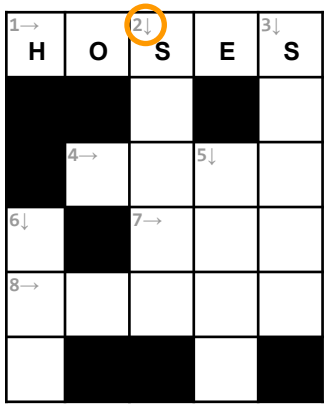
1 across val "hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which ones are used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```


2 down

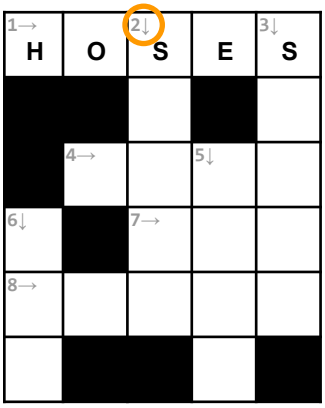
1 across val "hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

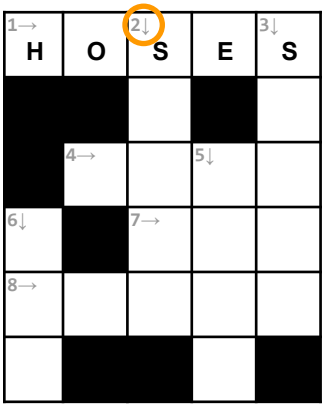
2 down		
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

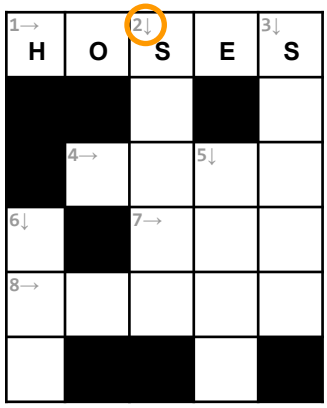
2 down	val	"aft"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Words not reused

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

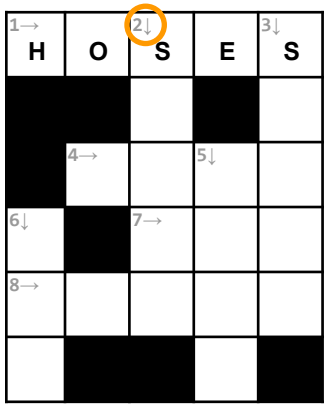
2 down	val	"ale"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Words not reused

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

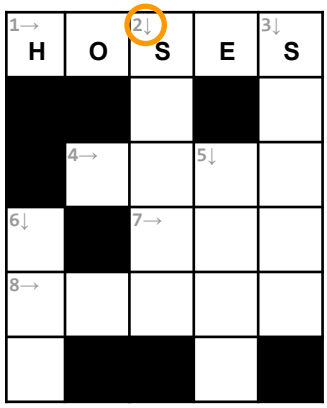

2 down	val	"eel"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which one is unused

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

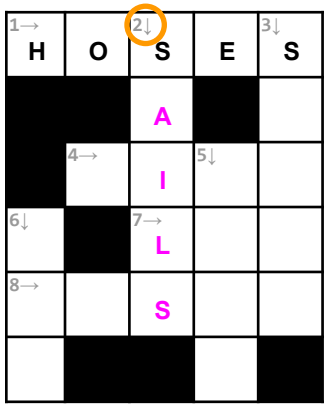
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which one is unused

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

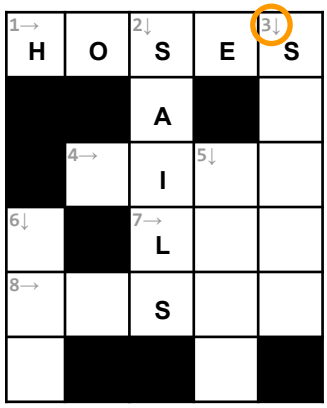
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which one is used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

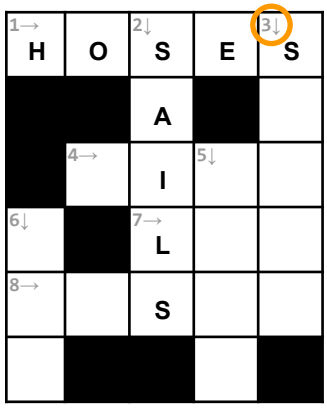
3 down		
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which one is used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

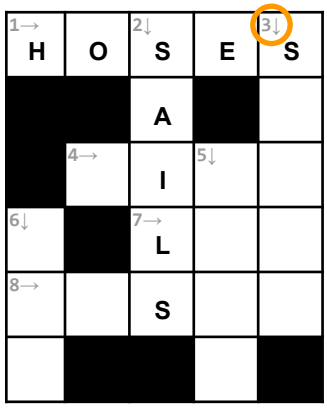

3 down	val	"aft"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

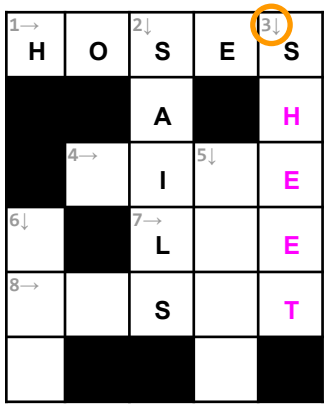
3 down	val	"ale"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

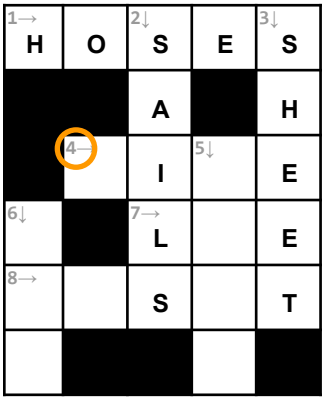

3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

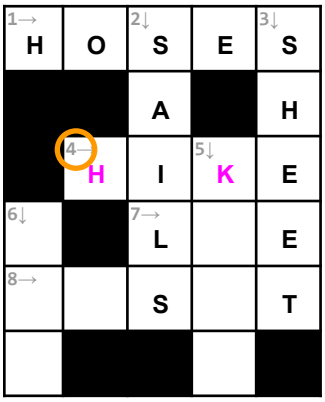
4 across		
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

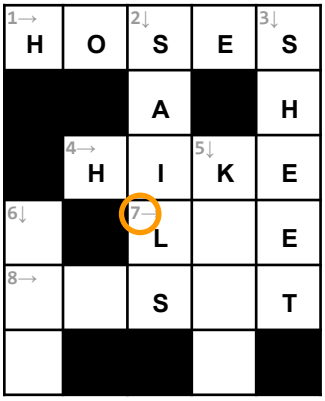
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

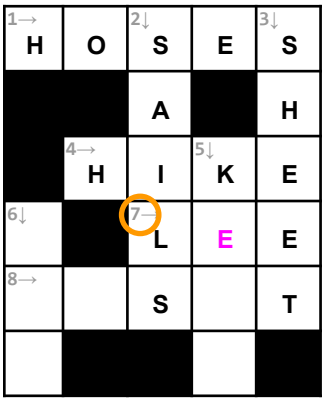
7 across		
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

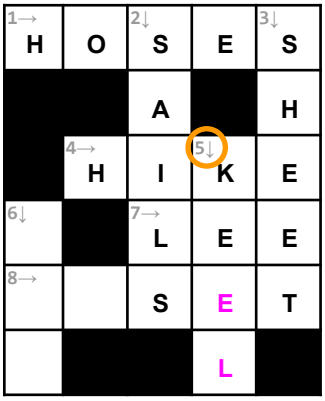
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Words can be used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

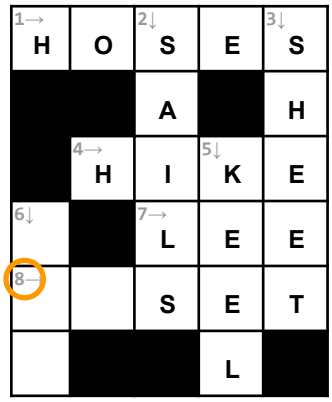
5 down	val	"keel"
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Words can be used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

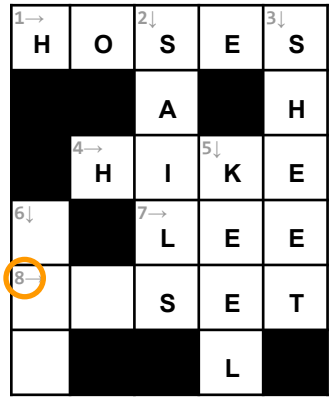

8 across		
5 down	val	"keel"
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



No words are valid! Must backtrack

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

8 across		
5 down	val	"keel"
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

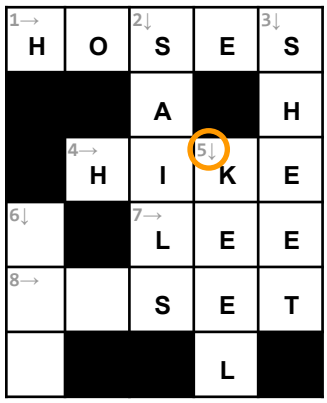
1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses

keel

laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

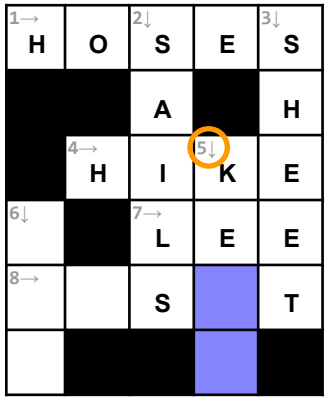
5 down	val	"keel"
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

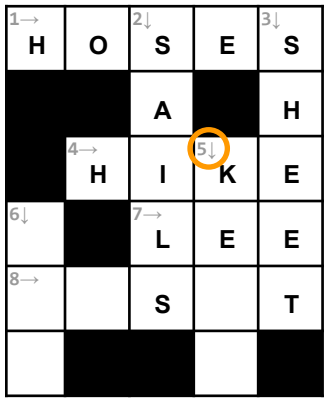
5 down		
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Which words used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

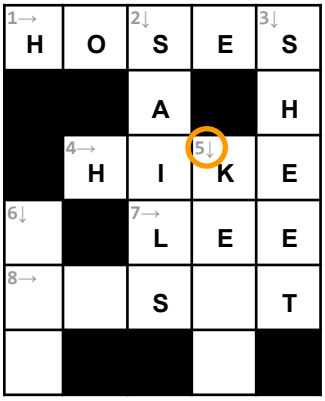
5 down		
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not reused

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



No words are valid! Must backtrack

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

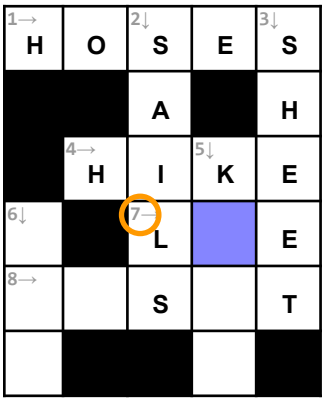
5 down		
7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not reused

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

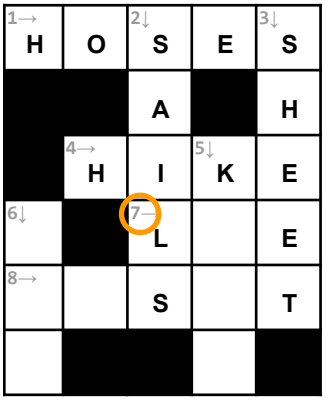

7 across	val	"lee"
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie



No words are valid! Must backtrack

```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

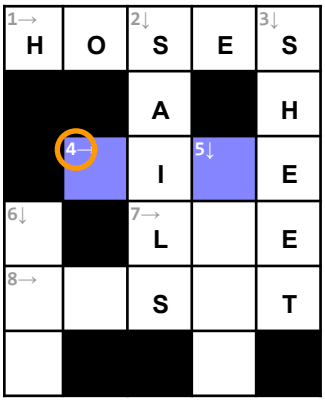
7 across		
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

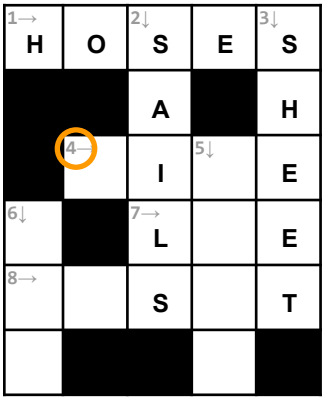
4 across	val	"hike"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

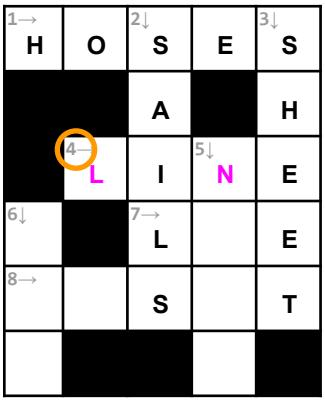
4 across		
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, other params)) {
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

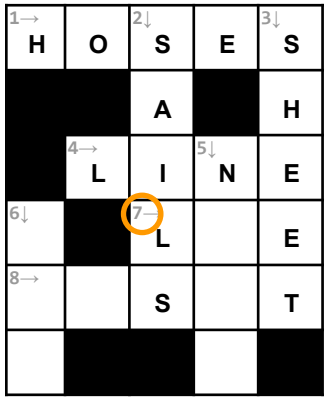

4 across	val	"line"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

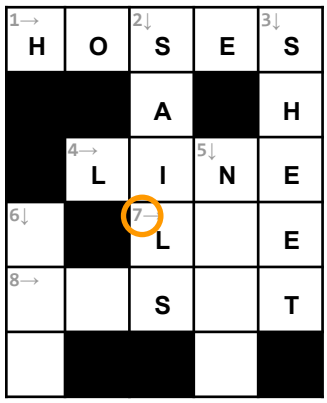
7 across		
4 across	val	"line"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not yet used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```


Notice that, since this is a fresh stack frame for n == 7, we start the loop over from the beginning again!

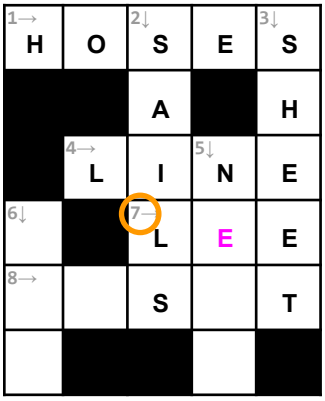
7 across		
4 across	val	"line"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```

Notice that, since this is a fresh stack frame for n == 7, we start the loop over from the beginning again!

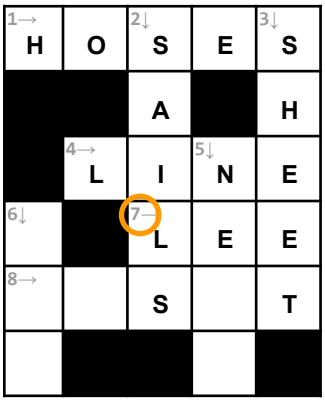
7 across	val	"lee"
4 across	val	"line"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"

Recursive Backtracking

1. Intersecting letters match
2. Must fit in slot (length)
3. Word not used

We consider another problem that can be solved with recursive backtracking. Given a list of words that are to be filled into a crossword puzzle, how do we find a solution that satisfies the rules of the conventional crossword?

aft
ale
eel
heel
hike
hoses
keel
laser
lee
line
sails
sheet
steer
tie




```
boolean findSolution(n, other params) {
    if (we found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val)) {
            applyValue(val);
            if (findSolution(n + 1, ot
                return true;
            }
            removeValue(val);
        }
    }

    return false;
}
```


And so on...

7 across	val	"lee"
4 across	val	"line"
3 down	val	"sheet"
2 down	val	"sails"
1 across	val	"hoses"