

think in terms of choosing where the first queen goes, then choosing where the second queen goes, and so on. There are 64 places to put the first queen because the chessboard is an 8-by-8 board. At the top of the tree, there are 64 different choices you could make for placing the first queen. Then once you've placed one queen, there are 63 squares left to choose from for the second queen, then 62 squares for the third queen, and so on.

Because backtracking searches all possibilities, it can take a long time to execute as there are many possibilities to explore. If we explore them all, we'll need to look at $64 * 63 * 62 * \dots * 57$ states, which is too many even for a fast computer. We need to be as smart as we can about the choices we explore. In the case of 8 queens, we can do better than to consider 64 choices followed by 63 choices followed by 62 choices, and so on. We know that most of these aren't worth exploring.

One approach is to observe that if there is any solution at all to this problem, then the solution will have exactly one queen in each row and exactly one queen in each column. That's because you can't have two in the same row or two in the same column and there are 8 of them on an 8-by-8 board. We can search more efficiently if we go row-by-row or column-by-column. It doesn't matter which choice we make, so let's explore column by column. Eliminating undesirable candidates from being explored is also called *pruning* the decision tree.


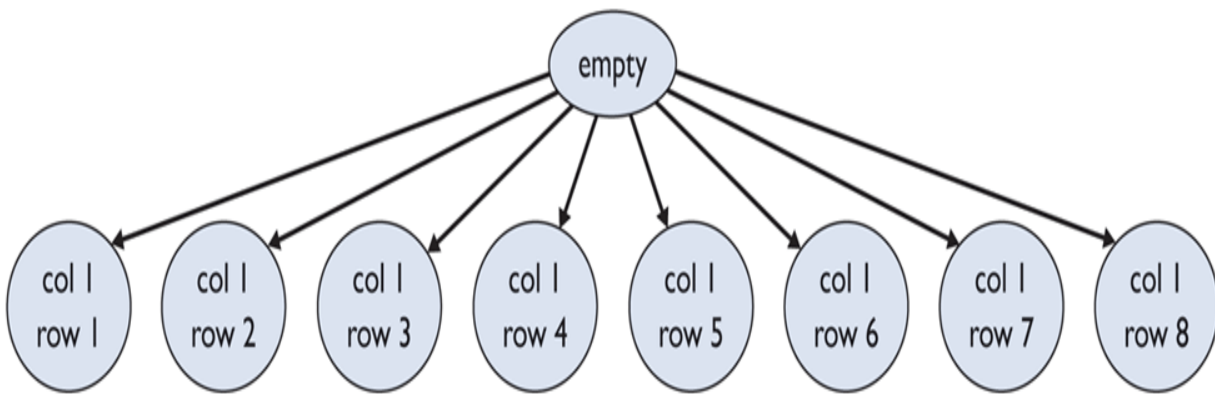
In this new way of looking at the search space, the first choice is for column 1. We have 8 different rows where we could put a queen in column 1. At the next level we consider all of the places to put a queen in column 2, and so on. **Figure 12.11**  shows the top of our decision tree.

Figure 12.11 Decision tree for first column




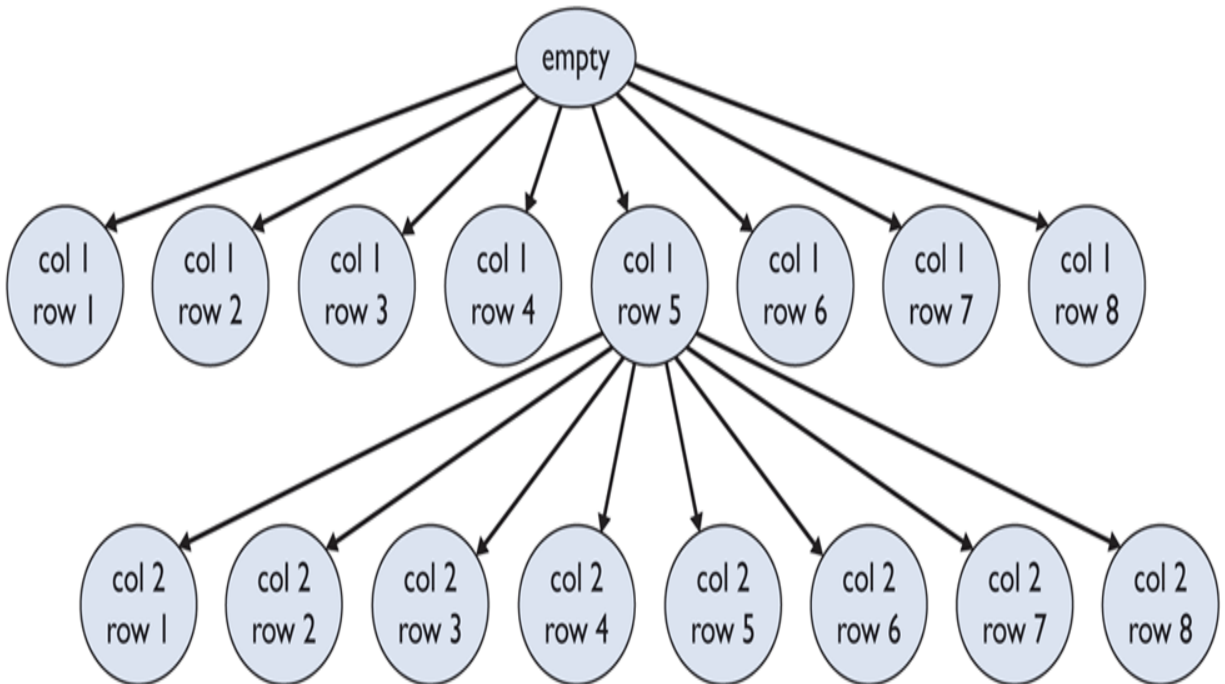
There are eight different branches for each column. Under each of these branches, we have eight branches for each of the possible rows where we might place a queen in column 2. For example, if we think just about the possibility of placing the first queen in column 1 and row 5 and then think about all of the ways to place a second queen, we end up with an extra level of the tree as shown in **Figure 12.12** .

Figure 12.12 A decision tree for second column



These pictures don't capture the whole story because the tree is so large. There are eight branches at the top. From each of these eight branches there are eight branches. And from each of those branches there are eight branches. This continues eight levels deep (one level for each column of the board).

It's clear that the eight choices could be coded fairly nicely in a `for` loop, something along the lines of:

```
for (int row = 1; row <= 8; row++) {
```

But what we need for backtracking is something more like a deeply nested `for` loop:

```

for (int row = 1; row <= 8; row++) { // explore column 1
    for (int row = 1; row <= 8; row++) { // explore column 2
        for (int row = 1; row <= 8; row++) { // explore column
3
            for (int row = 1; row <= 8; row++) { // explore
column 4
                ....

```

That's not a bad way to think of what backtracking does, although we will use recursion to write this in a more elegant way.

Before we explore the backtracking code, we need to consider the low-level details of how to keep track of a board that allows us to place queens in specific locations. It is helpful to split off the low-level details into a separate class. Let's plan on writing a `Board` class that allows us to construct a `Board` object to keep track of the state of the chessboard.

What kind of methods would we want to have for a `Board` object? Obviously it would need some kind of constructor. We want to pass it an integer n so we could solve the more general " n queens" problem with an n -by- n board. We need to be able to test whether it's safe to place a queen at a particular location. We need a way to place a queen on the board. We need a way to remove a queen because the backtracking involves trying different possibilities. We need a way to display output showing where the queens have been placed. Lastly, we'd like to be able to ask the board what size it is.

These can be implemented as the following constructor and methods:

```
public Board(int size)
public boolean isSafe(int row, int column)
public void place(int row, int column)
public void remove(int row, int column)
public void print()
public int size()
```

Let's assume that we have a `Board` class that implements all of these methods. That's not the interesting code. The interesting code is the backtracking code which, given this class, we can now write in a very straightforward manner.

Once again we will introduce a public/private pair of methods to perform the backtracking. The public method can be passed a `Board` object and it will call the private recursive method that performs the backtracking:

```
public static void solve(Board b) {
    ...
}
```

Recall that we had this general pseudocode for backtracking solutions:

```
private static void explore(a scenario) {  
    if (this is a solution) {  
        report it.  
    } else if (this is not a dead-end) {  
        use recursive calls to explore each available choice.  
    }  
}
```

The 8-queens backtracking problem differs in several important ways from the simple backtracking we saw before. Let's consider each of the differences and see how to adapt our pseudocode for each of them.

In the traveling problem we had just three possibilities to consider, so it made sense to write three recursive calls to explore each possibility. Here we have eight possibilities for the 8-queens problem and potentially a different number of possibilities if the board size is something other than 8. In this case, we want to use a loop to consider the different possibilities. Many backtracking problems will require using a loop instead of individual calls; it is useful to adapt our pseudocode to fit that approach.

```
private static void explore(a scenario) {  
    if (this is a solution) {  
        report it.  
    } else {
```

```
    for (each available choice) {  
        if (this is not a dead-end) {  
            use a recursive call to explore that choice.  
        }  
    }  
}
```

We can also be more specific about what it means to explore a choice. In our simple example, we were building up a string that stored the path. That meant that we didn't have to undo a choice to move on to the next choice. More complex backtracking problems require a cleanup step where you undo a choice. That will be true of 8-queens. We will place a queen on the board to explore that branch and when we come back from the recursive exploration, we will need to remove the queen to get ready to explore the next possible choice. So the pseudocode can be expanded to include the pattern of making a choice, recursively exploring, and then undoing the choice:

```
private static void explore(a scenario) {  
    if (this is a solution) {  
        report it.  
    } else {  
        for (each available choice) {  
            if (this is not a dead-end) {  
                make the choice.  
            }  
        }  
    }  
}
```

```
        recursively explore subsequent choices.  
    undo the choice.  
}  
}  
}  
}
```

This pseudocode is general enough that it can be used for many backtracking problems. Some programmers refer to this as the “choose, explore, un-choose” pattern for backtracking.

This is appropriate code to use if you want to search all possibilities. In the case of 8-queens, there are many solutions (over 90 different solutions). We don’t really want to see all of these solutions. We’re happy to have just one. For this backtracking problem, we will consider a variation that stops when it finds a solution. That means that our recursive method will need to have a way to let us know whether a certain path worked out or whether it turned out to be a dead-end. A good way to do this is to have a `boolean` return type and to have the method return `true` if it succeeds, `false` if it is a dead-end.

```
private static boolean explore(a scenario) {  
    if (this is a solution) {  
        report it.  
        return true;  
    } else {
```



```

    for (each available choice) {
        if (this is not a dead-end) {
            make the choice.
            if (recursive call to explore subsequent
choices) {
                return true;
            }
            undo the choice.
        }
    }
    return false;
}

```

This pseudocode is also general enough that it can be used for many backtracking problems when you want to stop the process after finding a solution.

We can adapt this to the 8-queens problem by filling in the details. What parameters will it need to specify a scenario? It certainly needs the Board object. Recall that each level of the decision tree involves a different column of the board. The first invocation will handle column 1, the second will handle column 2, and so on. Therefore, in addition to the board, the method also needs to know the column to work on. That leaves us with this header:

```

private static boolean explore(Board b, int col) {
    ...
}

```

```
}
```

How do we know if we've found a solution? This backtracking code doesn't explore dead-ends. As a result, it has the following precondition:

```
// pre: queens have been safely placed in previous columns
```

What would be a nice column to get to? A tempting answer is 8. It would be nice to get to column 8 because it would mean that 7 of the 8 queens have been placed properly. But an even better answer is 9, because the precondition tells us that if we ever reach column 9, then queens have been safely placed in each of the first 8 columns.

This turns out to be our test for whether we have found a solution:

```
private static boolean explore(Board b, int col) {  
    if (col > b.size()) {  
        return true;  
    } else {  
        ...  
    }  
}
```

The pseudocode indicates that we should print the answer before returning true. We could do that, but because the solution is stored in the `Board` object, we can simply return and allow the calling method to print out the solution.

Now we have to fill in the details of the `for` loop that explores the various possibilities. We have eight possibilities to explore (the eight rows of this column where we might place a queen). A `for` loop works nicely to explore the different row numbers. The pseudocode indicates that we should test to make sure it is not a dead-end. We can do that by making sure that it is safe to place a queen in that row and column:

```
for (int row = 1; row <= b.size(); row++) {  
    if (b.isSafe(row, col)) {  
        ...  
    }  
}
```

We now need to fill in the three steps that are involved in exploring a choice: making the choice, recursively exploring subsequent choices, and undoing the choice. We make the choice by telling the board to place a queen in that row and column:

```
for (int row = 1; row <= b.size(); row++) {  
    if (b.isSafe(row, col)) {
```

```
        b.place(row, col);  
        ...  
    }  
}
```

Then we recursively explore subsequent choices (later columns) and return true to stop the process if it finds a solution:

```
for (int row = 1; row <= b.size(); row++) {  
    if (b.isSafe(row, col)) {  
        b.place(row, col);  
        if (explore(b, col + 1)) {  
            return true;  
        }  
        ...  
    }  
}
```

Finally, we have to undo our choice in case that turns out to be a dead-end:

```
for (int row = 1; row <= b.size(); row++) {  
    if (b.isSafe(row, col)) {  
        b.place(row, col);  
        if (explore(b, col + 1)) {  
            return true;  
        }  
    }  
}
```

```

    }
    b.remove(row, col);
}
}

```

Putting it all together we get:

```

private static boolean explore(Board b, int col) {
    if (col > b.size()) {
        return true;
    } else {
        for (int row = 1; row <= b.size(); row++) {
            if (b.isSafe(row, col)) {
                b.place(row, col);
                if (explore(b, col + 1)) {
                    return true;
                }
                b.remove(row, col);
            }
        }
        return false;
    }
}

```

We need some code in the `solve` method that starts the recursion in column 1 and that either prints the solution or a message about there

not being solutions:

```
public static void solve(Board solution) {  
    if (explore(solution, 1)) {  
        System.out.println("One solution is as follows:");  
        solution.print();  
    } else {  
        System.out.println("No solution.");  
    }  
}
```

On our web site <http://www.buildingjavaprograms.com/> you will find the `Board` class and a full runnable version of the program that you can run to see that it finds a solution to 8 queens, 4 queens, and so on. In the case of 2 queens, there are no solutions. Our graphical variation of the program includes an animation generated by the backtracking itself.

Solving Sudoku Puzzles

Let's look at one more example of backtracking. Sudoku puzzles have become very popular. The puzzle involves a 9-by-9 grid that is to be filled in with the digits 1 through 9. Each row and column is required to have exactly one occurrence of each of the nine digits. The grid is further divided into nine 3-by-3 grids, and each grid is

also required to have exactly one occurrence of each of the nine digits. A specific Sudoku puzzle will fill in some of the cells of the grid with specific digits, such as the puzzle shown in [Figure 12.13](#).

Figure 12.13 A Sudoku board

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

When people play Sudoku, they use all sorts of heuristics and intuitions to find a way to fill in the empty cells of the grid while still meeting the constraints. With a backtracking approach, we just use brute force to try all possibilities.

The choices involved are what digit to put in each of the unoccupied cells of the grid. There are nine possibilities to explore for each empty cell (the digits 1 through 9). This approach to Sudoku is not necessarily going to run quickly, but backtracking provides a framework for quickly writing a complete solution.

As with many backtracking tasks, the real work involves keeping track of the details of this particular scenario. With the 8-queens problem we introduced a `Board` class to keep track of the chessboard. In this case we should develop a `Grid` class that keeps track of the state of the Sudoku grid.

In the case of the Sudoku grid, we have to start with some of the cells filled in. That will mean that our `Grid` class will need to read the initial configuration. Let's assume that each puzzle is stored in a text file with a 9-by-9 grid of numbers and dashes, as in:

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


That means that our `Grid` class constructor will need to read the initial configuration:

```
public Grid(Scanner input)
```

We immediately run into another problem. If some of the cells are filled in, then it's not as simple as the 8-queens case where we could just systematically explore each different cell of an empty board. In the Sudoku case, we have to figure out which cell to explore first, which cell to go to after that, and so on. There are many ways we could approach this, but the simplest is to give this responsibility to the `Grid` class. We can introduce a method that allows us to ask the grid to give us the next unassigned location in the grid.

Then we are faced with the question of how to specify a grid location. An obvious approach would be to give row and column numbers, but a method can return only one value. If the grid is going to identify the unassigned locations, then we don't really need to keep track of rows and columns ourselves. To keep thing simple, let's assume that the grid returns a cell number as a simple `int`:

```
public int getUnassignedLocation()
```

Eventually the grid will fill up, so we have to adopt a convention for what happens when there is no unassigned location left. Let's

assume that the method returns `-1` in that case to indicate that the grid is full.

In addition to these methods, we will want similar methods to the ones we had in 8-queens. We want to be able to test whether it's safe to set a particular cell to a particular digit. This is the method that will make sure that the grid we are building up is legal. So it will have to check to see if this new digit doesn't already appear in the given row, column, or subgrid. We'll need a method to set a cell to a specific digit. An "undo" method to remove a digit from a cell will be necessary. We will also want a method to print our solution. These are implemented as the following methods, respectively:

```
public boolean noConflicts(int cellNumber, int n)
public void place(int cellNumber, int n)
public void remove(int cellNumber)
public void print()
```

As with 8-queens, we'll want to stop searching once we find a solution, so we start with the standard pseudocode for the single-solution backtracking:

```
private static boolean explore(a scenario) {
    if (this is a solution) {
        report it.
        return true;
    }
```