Backtracking

See Section 7.7 p 333-336 of Weiss

So far we have looked at recursion in general, looked at Dynamic Programming as a way to make redundant recursions less inefficient, and noticed that recursion can be simulated with a loop and a stack.

We will now talk about a technique called *Backtracking* that uses recursion to solve certain types of problems.

The kinds of problems that can be addressed with backtracking have solutions that can be built up with a sequence of steps, where we can enumerate the possible choices for each step.

For example, there is a classic question about finding n squares on an nxn grid so that no two are in the same row, the same column, or lie on the same diagonal. This is called the "n-Queens" problem because you can think of it as putting n Queen pieces on an nxn chessboard without any of them threatening the others. This problem has n steps -- choosing a row for the first column, a row for the second column, etc. Each of those steps has n choices -- use row 0, row 1, and so forth.

Sudoku puzzles can be solved by backtracking. There are 81 steps (well, 81 minus the given numbers). The possible choices for each step are the numbers 1 through 9.

Finding a path through a maze can be solved by backtracking. At each step the possible choices are to move left, right, straight ahead, or straight back.

With the previous problems we are looking for any one solution. Sometimes we want to find all solutions. For example, we might want to find all of the permutations of the numbers 0..n-1. This can be done with backtracking -- the jth step is finding the jth entry of the permutation; the possible candidates are the numbers 0..n-1.

Sometimes backtracking can be used to solve minimization problems. For example, we might want to find the cheapest ordering of a set of cities, where the cost is the cost of flying from one to the next. We can do this by finding all orderings of the cities, measuring the cost of each, and keeping track of the minimum cost.

Weiss gives an example of using backtracking to find the next move in a TicTacToe game. The idea here is to use backtracking to generate all possible situations that might result from the current state of the game-- first the computer will move, then the opponent, then the computer and so forth. We then treat this as an optimization problem -- maximizing the benefit (to the computer) and minimizing the benefit to the opponent.

We will first consider the sort of problem where we are looking for any one solution. Here is pseudocode; this assumes that the solution has n steps and that we can record the solution in an array of base-type E.

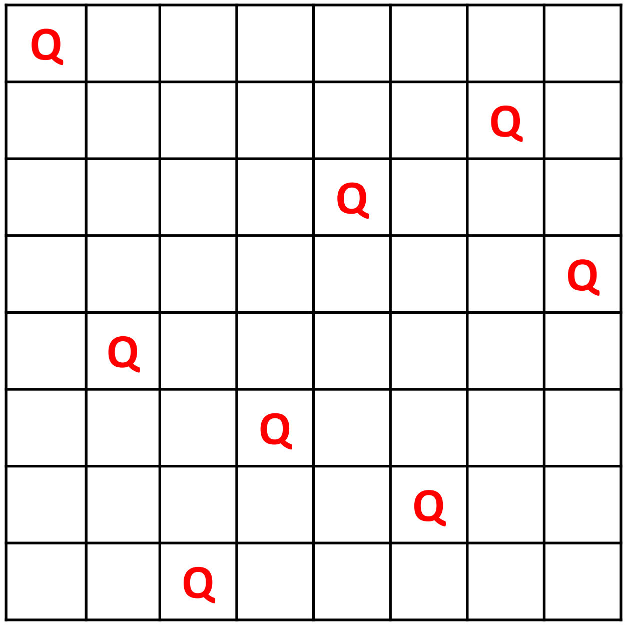
The signature of the backtracking method is

```
public boolean solve(int i, E[] Solution)
```

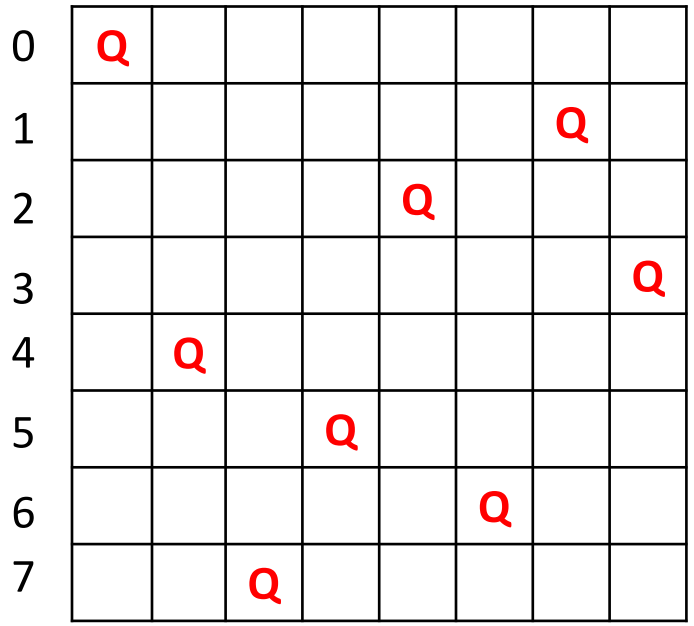
This assumes we have already filled in the entries of the Solution array for indices [0] through [i-1] and that these are compatible with a solution. The method ties all of the possible values for step i, and with each recurses to solve the problem for i+1

```
public boolean solve(int i, E[] Solution) {
        if (i == n)  // i.e if we are done {
                print or do something with the solution
                return true;
        }
        else {
                for each of the possible values j for step i {
                        Solution[i] = j;
                        if (entries of Solution are compatible)
                                if (solve(i+1, Solution))
                                        return true;
                }
                return false;
        }
}
```

This has an easy application to the n-Queens problem. We want to choose n squares on an nxn board so that no two are in the same row, column or diagonal:



Our backtracking algorithm will choose one row for each column and put these rows into an array I call Rows.



For this picture the Rows array is **[0, 4, 7, 5, 2, 6, 1, 3]**

Here is the backtracking method that finds solutions to the problem:

```
public static boolean solve(int i, int[] Rows) {
        if (i == Rows.length)
                return true;  // the caller prints the solution
        else for (int j = 0; j < Rows.length; j++) {
                Rows[i] = j;
                if (Ok(Rows, i))
                        if (solve(i+1, Rows))
                                return true;
        }
        return false;
}
```

The Ok(int[ ] Rows, int i) method checks that the entries of Rows from index 0 to index i are all compatible -- no two are in the same row, column or diagonal. We don't need to check on the columns because we only assign one row to each column.

To check that no two are in the same row, we just need to make sure that entries of Rows are all different numbers. One way to do this is to make an array of booleans, which Java initializes to False, and to make each entry corresponding to a row be true:

```
public static boolean RowsOk(int[] A, int i) {
        boolean[] used = new boolean[A.length];
        for (int j = 0; j <= i; j++) {
                if (used[A[j]])
                        return false;
                else
                        used[A[j]] = true;
        }
        return true;
}
```

For the diagonals we need to think about the indices as we move along a diagonal.

row=0, col=0	row=0, col= 1	row=0, col=2	row=0, col=3
row=1, col=0	row=1, col=1	row=1, col=2	row=1, col=3
row=2, col=0	row=2, col=1	row=2, col=2	row=2, col=3
row=3, col=0	row=3, col=1	row=3, col=2	row=3, col=3

Consider first the diagonals that move upward from left to right. The row and column indices on such a diagonal all sum to the same value. For example, there is a diagonal consisting of the (2, 0), (1, 1) and (0, 2) entries. All of these entries have their row and column sum to 2. The Rows array has its upward diagonals okay if no two entries have their row and column sum to the same value

```
public static boolean UpDiagOk(int[] A, int i) {
        boolean[] used = new boolean[2*A.length];
        for (int j = 0; j <= i; j++) {
                int sum = j+A[j];
                if (used[sum])
                        return false;
                else
                        used[sum] = true;
        }
        return true;
}
```

The downward diagonals have the difference between their row and column constant. For example, one such diagonal has entries (1, 0), (2, 1), and (3, 2). These indices all differ by 1. We can use this to write a DownDiagOk() method.

Altogether, the Ok(int[] Rows, int i) method for the Queens problem is

```
public boolean Ok(int Rows, int i ) {
        return RowsOk(Rows, i) && UpDiagOk(Rows, i) && DownDiagOk(Rows, i);
}
```

Our basic backtracking paradigm quits as soon as it finds one solution:

```
public boolean solve(int i, E[] Solution) {
    if (i == n)  // i.e if we are done
        print or do something with the solution
    else {
        for each of the possible values j for step i {
            Solution[i] = j;
            if (entries of Solution are compatible)
                if (solve(i+1, Solution))
                    return true;
        }
        return false;
    }
}
```

If instead of one solution we want to generate all possible solutions, there is no longer a need to return true or false; we always run through all candidates:

```
public  void solve(int i, E[] Solution) {
        if (i == n)  // i.e if we are done
                print or do something with the solution
        else {
                for each of the possible values j for step i {
                        Solution[i] = j;
                        if (entries of Solution are compatible)
                                solve(i+1, Solution);
                }
        }
}
```

Here is how this applies to finding all solutions of the n-Queens problem:

```
public static void allSolutions(int i, int[] Rows) {
        if (i == Rows.length) {
                printBoard(Rows);
        }
        else for (int j = 0; j < Rows.length; j++) {
                Rows[i] = j;
                if (Ok(Rows, i))
                        allSolutions(i+1, Rows);
        }
}
```

Now write a function that prints all permutations of the numbers from 0 to n-1.