COMP 333
N Queens in Prolog

The Prolog solution to N Queens is simpler in some ways than the Java solution since the Prolog solution only needs to describe the logical relationships between the column positions, and does not have to do any bookkeeping or backtracking.

There are many approaches, I'll describe some of the basic ones here.

For any approach we will need a way to represent the set of queen positions and a way to confirm no conflicts for positions in any two columns.

Number of Ways to Position Queens
Before looking at ways to search for solutions, let's count how many ways there are to place 8 queens on an 8-by-8 board. In the most general case, without imposing any constraints, without eliminating any positions based on knowledge of the solutions, we get

# of positions = 64*63*62*61*60*59*58*57 = 178,462,987,637,760 or about 178 trillion

We get this number by noticing that the 1st queen can be placed on any of 64 squares, the next on any of 63 squares, and so on. We could search for solutions by considering each of these possible solutions and eliminating any case if it contains column, row, or diagonal conflicts. But as a practical matter, we don't really need to consider all of these positions. We can eliminate most by noticing some general facts about the form of solutions.

We can eliminate all positions with column conflicts by counting only positions with one queen per column, which gives

# of positions with no column conflicts = 8^8 = 16,777,216 or about 16 million

This follows because for each column there are 8 possible positions, and each column position is independent from the others.

We can further reduce the positions to consider by also eliminating those with row conflicts. This means we only consider solutions that are some permutation of the values {0,1,2,3,4,5,6,7}. The number of permutations of a set of 8 items is 8!

# of positions with no row or column conflicts = 8! = 8*7*6*5*4*3*2 = 40,320

This follows by noting that any solution for the 8-by-8 case will have one position per column and each position must be different from any other column, leaving us with a permutation of the given list.

In the two approaches described above, the first is based on generating permutations. Even though it doesn't seem that efficient, it only has to consider about 40 thousand positions. In the second, we build the solution one column at a time, which seems more direct than generating permutations. But the column-by-column approach has about 16 million positions to consider.

No matter the approach, we already know that out of all the possible positions, only 92 are actually solutions.

Solution Search #1: Permutations of List of Unique Values

Use a list containing the numbers 0 through N-1 as a list of queen positions. Indexes will represent columns and the value in the list at that index will represent the row position for the queen in that column. For N=8, you can use

```
permutation([0,1,2,3,4,5,6,7],X).
```

We know from the discussion above that there are about 40 thousand possible values of X. Prolog will try each of them via backtracking.

For a general value of N, you will first need to define a predicate that constructs a list of values from 0 through N-1, then take the permutation of that list.

Define a predicate nc/3 that confirms that a specific column pair have no conflicts for a specific list of positions:

```
nc(L,A,B).
L:  list of queen positions
A,B:  indexes into list L, assume 0 <= A < B < length(L)
```

Note that this predicate is not recursive and has no special cases or additional clauses. Also note that nc stands for “no conflicts” so a result of **true** means “no conflicts” and **false** means “there are conflicts”.

Brute Force Solution
We can obtain brute-force solutions for the N=8 case by using permutation/2 and creating a pool of all 40K possible position lists, then calling nc/3 for every column pair:

```
bruteforce(Y) :-
        permutation([0,1,2,3,4,5,6,7],Y),          % let permutation/2 find a potential solution Y
        nc(Y,0,1),                                 % check Y for conflicts for all column pairs
        nc(Y,0,2),nc(Y,1,2),
        nc(Y,0,3),nc(Y,1,3),nc(Y,2,3),
        nc(Y,0,4),nc(Y,1,4),nc(Y,2,4),nc(Y,3,4),
        nc(Y,0,5),nc(Y,1,5),nc(Y,2,5),nc(Y,3,5),nc(Y,4,5),
        nc(Y,0,6),nc(Y,1,6),nc(Y,2,6),nc(Y,3,6),nc(Y,4,6),nc(Y,5,6),
        nc(Y,0,7),nc(Y,1,7),nc(Y,2,7),nc(Y,3,7),nc(Y,4,7),nc(Y,5,7),nc(Y,6,7).
```

Although this is not a good solution, it again illustrates Prolog's backtracking capabilities. Potential solutions are found in goal #1 using permutation/2. Then the potential solution is checked against nc/3 for all indicated combinations of columns until some goal evaluates to false (conflict found). If a conflict is found at any point, that potential solution is abandoned, and backtracking returns to the first goal to try a different solution, continuing until a solution is found that satisfies all the goals (no conflicts).

I recommend you try this solution just as an experiment to see that it works and to help understand the problem. But you won't turn this in as your final program, we'll write something a little more compact as the official version.

We can somewhat improve the solution by introducing a counter predicate that covers all the required column comparisons without having to enumerate them individually.

One Column At A Time (OCT) Solution

The previous solution is similar to the sorting example where we created a Prolog predicate that used the predicates randlist/3, permutation/2, and is_sorted/1 to create the appearance of performing a sort. We can do the same thing for 8 Queens where we let Prolog try every possible solution, then for any solution confirm that no pairs of columns have conflicts. If any pair generate a conflict, we don't attempt to fix or adjust the solution, we just discard it (via backtracking) and try our luck with another one.

A different approach is to **build a solution one column at a time**, similar to how we find solutions in the Java version. In this approach, we start with an arbitrary value between 0 and n-1 that represents the queen position for a single column (index = 0 representing column 0). To get our initial solution list we create a list of length 1 containing that value.

```
NM1 is N-1,
between(0,NM1,P),           % P is a row position between 0 and N-1
L = [P].                    % L is an initial solution of length 1
```

We then generate a new column value and attempt to append it to the original list. In order to successfully append, the new value must generate no conflicts with any values already in the list. If there are no such values, then Prolog will backtrack to an earlier operation, undo one or more of the previously added positions, and add different positions and then move forward to find a solution with no conflicts.

```
between(0,NM1,P1),          % P1 is another row position between 0 and N-1
append([P1],L,L2),          % append P1 to head of L yielding L2 = [P1,P]
nc(L2).                     % check L2 for no conflicts
                            % if nc/1 fails due to conflicts, we backtrack to get another P1
```

Hard to say which solution is "better". The permutation solution seems primitive but starts with a quite small set of cases to consider, so its performance should be good. The OCT solution may better express the fundamental logic of the column constraints, but it depends on a much larger set of possible positions, so exploring them may take more time.

Predicates for OCT
Unlike the permutation-based search, the OCT search uses lists that start off with length 1 and grow to length n as more column positions are established.

We need two predicates with multiple parameter versions:

nc/1
nc/3
These predicates check a given list of positions for no conflicts.

solve/2
solve/3
These predicates build the solution one column at a time, which is similar to how we are doing it in Java.

nc/1 and nc/3
The nc/1 and nc/3 predicates evaluate to true for "no conflicts" for the indicated list. These predicates do not compute or construct any solution, they simply evaluate to true or false.

nc/1: nc(L) evaluates to true if list L contains no conflicts. This implies that all possible column pairs have been checked. In the definitions below, we achieve the confirmation through recursion on smaller and smaller lists. At any point in the recursion, we assume that the tail of the list has already been checked, and we now attempt to grow the list by suggesting a new position for the head of the list, checking it against all positions in the current list, and appending it at the head if it checks out, thereby growing the list by one position.

```
% nc/1
nc([]).                 % base case:  empty list has no conflicts
nc([X]).                % base case:  list with one column has no conflicts
nc([H|T]) :-            % nc/1 calls nc/3 to do the rest of the work
        nc(H,T,1).      % H:  new position; T: existing list;
                        % 1: distance (# of cols) between H and T

% nc/3
nc(_,[],_).             % base case:  no conflicts if param #2 is empty
nc(P,L,D) :- …          % general case:  P:  new position; L: list to check;
                        % D: distance (# of cols) between P and L

        % Checks for no conflicts between position P and all values in list L
        % Because we will break L apart into sublists, we must remember
        %       the distance between P and the head of list L
        % Initially, P and L are [P|L], so D = 1; as recursion continues, D is replaced by D+1
        %
        % When L is reduced to [], 1st rule is stopping condition
        %
        % Algorithm:
        %      check for no conflicts between P and head of L
        %      then call nc recursively on tail of L with P unchanged and D replaced with D+1
```

solve/2 and solve/3

The solve/2 and solve/3 predicates put the list together one column at a time and use nc/1 to check that the resulting list has no conflicts.

```
% solve/2:  N is dimension of board (e.g. 8) and X is the resulting solution list
solve(N,X) :-
        NM1 is N-1,
        between(0,NM1,P),           % between is built-in; picks position P betw 0 and N-1
        solve(N,[P],X).             % creates a list containing P and calls solve/3
                                    % N is dimension, 2nd param is temporary solution,
                                    % 3rd param is final solution

solve(N,X,X) :-                     % stopping rule:  when length of 2nd param X reaches N,
                                    % 3rd param (solution) = 2nd param (temporary solution)
        length(X,XL),
        XL = N.

solve(N,T,X) :-   % N:  dimension; T: temporary solution; X: final solution
                  % Algorithm:
                  %   pick another value P between 0 and N-1
                  %   append it to the front of list T yielding T2
                  %   check that T2 has no conflicts
                  %   call solve/3 recursively with T replaced with T2, N and X unchanged
                  %   when length of list T reaches N, previous rule will recognize
                  %   stopping condition
```