COMP 333
N Queens Problem in Racket and Prolog

The N Queens problem is a famous mathematical puzzle first described in an 1848 paper as the 8 Queens problem, since generalized to N Queens. Given a chess board of size n-by-n, and given 8 queens (color or player not important), a solution to the problem is any positioning of all n queens such that no queen attacks another queen. This is equivalent to one queen per row, one queen per column, and one queen per diagonal.

There is no effective general algorithm. In other words, the only way to find solutions is to try all possible arrangements and check each one to see if it satisfies the constraints. Solutions are found by brute-force search that try every possible positioning. For each positioning tried, if the constraints are satisfied, then it is a solution. If the constraints are not satisfied, then try another position. Search continues until all possible solutions are tried.

A successful search requires that proper bookkeeping is done to ensure that (1) positions are only considered once, and (2) no positions are omitted. If (1) is violated, the search may enter an infinite loop, trying the same positions over and over. If (2) is violated, then the search may not find a solution, since the solution might be the position that was never considered. When coding a solution in a language like Java or even Racket, checking a position to see if it satisfies the constraints is simple. What's harder is having a proper plan to generate every position and only consider it once. As we will see shortly, coding the solution in Prolog will be substantially simpler because Prolog handles most details of the bookkeeping of positions automatically through its built-in search and backtracking engine.

The puzzle predates computer programming, but it is now a popular puzzle to solve with programming. It's also good practice for considering approaches to other constraint checking problems where a large number of potential solutions must be considered. Knowing N Queens may help you later for similar problems that can be solved by analogy to this one.

How Much Work Does It Take?

Let's consider how many possible solutions have to be considered in the case of n = 8. The number depends on how much initial pruning of the set is desired. If we choose no pruning, then there are:

64 X 63 X 62 X 61 X 60 X 59 X 58 X 57 = 178,462,987,637,760 (178 trillion)

positionings. This is how many different ways there are to position 8 queens on an 8-by-8 board, with the entirely reasonable restriction of only one piece per square. If we restrict positions to one queen per column, then there are:

8 X 8 X 8 X 8 X 8 X 8 X 8 X 8 = 8^8 = 16,777,216 (16 million)

positionings. If we further restrict positions to 1 queen per row and column, then there are:

8 X 7 X 6 X 5 X 4 X 3 X 2 X 1 = 8! = 40,320 (40 thousand)

Two popular ways to search for solutions are (1) the **one-column-at-a-time** approach, which means there will be about 16M different positions, and (2) the **permutation** approach, where we create a vector with all unique row values [0,1,2,3, …] then consider only permutations of the original vector,

which means there are about 40K solutions to consider. We will implement the one-column-at-a-time solution in this project.

Some of the tradeoffs to consider are that in the column-based approach, only row and diagonal conflicts must be checked for each of the 16M positions. In the permutation-based approach, only diagonal conflicts must be checked for each of the 40K positions. But it may be surprising to find out later that the column-based approach may have better performance even though it must check more positions and constraints. The permutation-based approach has slower performance because of the work required to generate the permutations. Also, the permutation-based approach is simpler if all permutations are generated separately in advance, then checked individually. For a problem with large n, this will require more stack/heap resources to precompute a large set of permutations. What is needed is a permutation generator that generates one permutation at a time in some sequence, like Heap's algorithm. But this would take more work to implement. The column-based approach has no resource usage problem, we merely create a next-row vector with one entry per column to keep track of which rows have been checked so far for each column.

Racket Solution

We will represent the size of the problem as n, which is only given a specific value during testing. Code should only be written in terms of n, rather than a specific size like 8.

We will represent the solution as a list qp (for queen positions) which starts out as the list '(0) representing the positioning of the queen in column 0 at row 0. We will grow the solution one entry at a time until (length qp) reaches n.

In the Racket solution, we have to do bookkeeping to keep track of which rows have already been tried for each column. If we didn't, the search could turn into an infinite loop since we would try one positioning, decide it didn't work, move to other positionings, and eventually circle back to a previously rejected positioning, and so on. In Racket we will also have to recognize when our search has reached a dead end for a specific column, and initiate backtracking to go back to a previous column, change the queen position, then try to move forward again. We will see shortly that the same solution in Prolog will be simpler since the bookkeeping and backtracking is provided automatically by the Prolog interpreter and does not have to be implemented explicitly by the programmer.

Algorithm In General Terms

The solution to the problem will be represented by a list of length n labeled **qp** for **queen positions**. Columns are represented by index positions and row for some column is represented by the in the list at that column index. For example, for n=8, one of the solutions is

qp = '(3 1 6 2 5 7 4 0).

which we interpret as follows:

col 0, row 3 <= queen #1 position
col 1, row 1 <= queen #2 position
col 2, row 6 <= queen #3 position
col 3, row 2 <= queen #4 position
…

and so on to position all 8 queens on the board. Since it is easier in Racket to make modifications to the head of a list rather than the tail, we will actually refer to the column numbers in reverse order, starting

with column 0 on the right. This is different from the internal indexing used by Racket. But how we number the columns doesn't affect the solution, which doesn't depend on absolute values of column indexes. We need column info when checking diagonal conflicts, but here we only need relative distance between columns, not absolute column values.

One Column At A Time Solution

We can look at a few figures to understand how the search for solutions will progress. The 8-by-8 chess board looks like this:

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0								X
1								
2							X	
3								
4						X		
5								
6					X			
7								

Figure: sample chess board with X marking the position of queens in columns 0-3.

However, we don't need to represent the complete board, we only need to represent the solution list and the next row list, which are both one dimensional arrays.

qp: '(6 4 2 0 …) index by column, entry is row value of queen position for that column

nr: '(0 7 5 3 1 …) index by column, entry is next row to try for that column

Also, we will develop our solution column by column from right to left with logical column indexes numbered at 0 starting on the right (the opposite of the actual internal Racket indexes where 0 starts on the left). The qp and nr lists only need to hold enough values to represent the current partial solution. The qp list starts off with length 0, and grows as additional queen positions are found and added. A single solution is complete when qp reaches a length of n. The nr list will generally be one item longer than qp. It has an extra entry on the left end to track the row value for current column. We won't add a value for that column to the qp list until we finish the constraint checking for that column.

We will use "c" to represent the current column which is initialized to 0. When we find a row value for the current column, we increment c to advance to the next column. When we exhaust all available rows for the current column and need to backtrack, we decrement c to back up to the previous column. This is an important measure to notice while the solution progresses. The solution will go through phases where c is incremented multiple times as we determine more entries for the solution. Then it will be interrupted by phases where it reaches dead ends and must erase part of the solution, backtrack by decrementing c, return to an earlier column, try another row value, then try to enter another phase of forward progress. There will be hundreds of forward and backward phases before a c reaches the final column and a solution is found.

Here are a few steps in the search for the first qp solution for an initially empty board.

c: 0 current column is column 0
qp: '() qp list is initially empty
nr: '(0) nr list is initially set to row 0 for column 0

	7	6	5	4	3	2	1	0
0								
1								
2								
3								
4								
5								
6								
7								

- Add 0 to qp to represent row position 0 for column 0
- Increment index 0 of nr, the next row to try for column 0, from 0 to 1, since row 0 has been tried
- Cons 0 onto head of nr to represent the start of search for a row for column 1
- Increment c from 0 to 1

c: 1
qp: '(0)
nr: '(0 1)

	7	6	5	4	3	2	1	0
0								X
1								
2								
3								
4								
5								
6								
7								

- Search row for column 1 that has no row or diagonal conflict with column 0; we find row 2 for column 1; cons 2 onto qp for row position 2 for column 1
- Set nr for column 1 to 3, since 0, 1, and 2 have been tried
- Cons 0 onto nr to represent start of search for column 2
- Increment c from 1 to 2

c: 2
qp: '(2 0)
nr: '(0 3 1)

	7	6	5	4	3	2	1	0
0								X
1								
2							X	
3								
4								
5								
6								
7								

- Search row for column 2 that has no row or diagonal conflict with columns 0 and 1; we find row 4; add 4 to qp to represent row position 4 for column 2
- Set nr column 2 to 5, since 0-4 have been tried
- Cons 0 onto nr to represent start of search for column 3
- Increment c from 2 to 3

c: 3
qp: '(4 2 0)
nr: '(0 5 3 1)

	7	6	5	4	3	2	1	0
0								X
1								
2							X	
3								
4						X		
5								
6								
7								

And so on, to move across the columns, finding a position for each queen one column at a time. For each new column n, note that the new position must be checked for conflicts against all previous columns n-1 through 0.

Checking for Conflicts

When searching for an acceptable row value for the next column to add to the existing partial solution qp, we make the following assumptions:

- The current value of qp is already conflict free because each value was checked against all the other values earlier.
- To add a new column, we only have to check the new value for no conflicts against each element in qp.

Note that if we had used the permutation based solution, the first assumption here would not hold. In the permutation approach, each entry in qp, must be checked for conflicts against every other entry in qp, which would be more work than what is required here.

Since we are using the column approach, we only have to check for **row** and **diagonal conflicts**. A row conflict means that two entries are the same for two different columns. A diagonal conflict means that the separation between two columns c_a and c_b which we can write as $abs(c_b – c_a)$ is the same as difference in the corresponding row values r_a and r_b which we can write as $abs(r_b – r_a)$. In other words, there is a diagonal conflict between two columns if

$abs(c_b – c_a) = abs(r_b – r_a)$.

For example, look at the following 8-by-8 board with some sample queen positions marked by letters a-f. Each position has both a column and row coordinate that defines that position. “a” is at row 0 col 0, “b” is at row 4 col 1, etc.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	a							
1					e			
2			c					
3								f
4		b						
5								
6								
7				d				

Here are some example pairs of positions and how to check for diagonal conflict:

- a-b: col diff = 1-0 = 1, row diff = 4-0 = 4: **no diagonal conflict**
- a-c: col diff = 2-0 = 2, row diff = 2-0 = 2: **diagonal conflict** because 2 == 2
- b-d: col diff = 3-1 = 2, row diff = 7-4 = 3: **no diagonal conflict**
- b-e: col diff = 4-1 = 3, row diff = 4-1 =3: **diagonal conflict** because 3 == 3
- d-f: col diff = 7-3 = 4, row diff = 7-3 = 4: **diagonal conflict** because 4 == 4
- c-e: col diff 4-2 = 2, row diff = 2-1 = 1: **no diagonal conflict**

Below are the description of the **nc** predicate functions which stands for **no conflict**. Let’s keep the same logic polarity throughout to avoid confusion. The nc predicates return **true** to mean **no conflicts**, and false to mean **there are conflicts.**

```
(define (nc-1 a b d)
        …
)
```

The function **nc-1** is a direct check of two row values a and b separated by column distance d. It evaluates to true if **a is not equal to b** (no row conflict) AND **row difference is not equal to column difference** (no diagonal conflict). The body is a simple logic expression, no conditional, no helper, no recursion.

```
(define (nc v p)
        …
)
```

The nc function checks that a new row value v has no conflicts with any of the entries in list p. It calls helper function nc-h to introduce a new parameter for distance d, the column separation between v and the first element of p. Initially d = 1, but for each recursive call it will increase by 1.

```
(define (nc v p)                ; top level function
        (nc-h v p 1)            ; call helper function with distance d = 1
)
(define (nc-h v p d)            ; helper function with distance d
        …
)
```

The helper function uses **nc-1** and **recursion** to confirm for no conflicts between v and any element of p. If p is empty (base case), the result is #t, otherwise it calculates logical AND of nc-1 applied to v, (car p), and d, with the result of the recursive call on v, (cdr p), and (+ d 1).

Example Trace of nc

```
(nc 5 '(1 3 4))
        => (nc-h 5 '(1 3 4) 1)
                => (AND         (nc-1 5 1 1)    ; distance between 5 and 1 is 1
                                (nc-1 5 3 2)    ; distance between 5 and 3 is 2
                                (nc-1 5 4 3)    ; distance between 5 and 4 is 3
                )
```

The solve Function

The **solve** function is the entry point that starts the search for a set of solutions. It uses the **nc** conflict checking predicate described above. **(solve n)** is the top level function and it calls the helper function **(solve-h …)** with additional parameters needed to do the necessary bookkeeping. The complete definition for (solve n) is:

```
(define (solve n) (solve-h n 1 '(0) '(1 0) '()))
```

In this definition I have skipped ahead one step and initialized qp to '(0) and nr to '(1 0). I could also initialize qp to '() and nr to '(0) and let the first step within the solve problem add the first entry to qp.

Helper Function (solve-h)

(solve-h n c qp nr sv)
n: value of n, the size of the problem, doesn't change
c: current column index
qp: partial solution list of queen positions determined so far
nr: next row list showing next row to try for each column while searching for solution
sv: solution vector, a list of lists of all solutions found so far, initially empty

```
(solve-h n c qp nr sv)
        (cond
                …
        )
)
```

All the back-and-forth search for solutions happens inside the conditional of **solve-h**. The conditions are:

Found one solution: (= c n): if current column c reaches n, then forward progress has advanced to column n, which is beyond the end of the board. This means that we have successfully positioned n queens in columns 0 through n-1 and qp constitutes one complete solution. We add current solution in qp to our solution vector sv, then backtrack by one column to start the search for the next solution. For example, in the case of n=8, there will be 16M positions to search, and out of those we will find 92 that are solutions.

End of search, all solutions found: (and (= (car nr) n) (empty? qp)): If this expression is true, the next row to try for the current column is the first entry in the nr list, which is (car nr). If (car nr) equals n, then there are no more rows available to try, we have tried them all. By itself this is a signal to backtrack from column c to c-1. But if in addition the solution list qp is also empty, then there is no earlier column to backtrack to, so we have reached the end of the entire search. The solve functions ends with return value sv, the solution vector or list of lists of all solutions.

Current column exhausted, backtrack to previous column: (= (car nr) n): if (= (car nr) n) but qp is **not** empty, this means that we have exhausted all possible row values for the current column but we are not yet done with the entire search. Here we backtrack to the previous column. We decrement c, we replace

qp with (cdr qp) to delete one column from the current solution, and we replace nr with (cdr nr) to erase the bookkeeping for the current column. We now try to find another row value for the column just deleted and try to move forward again.

Found row for current column, add it to partial solution, advance to next column: (nc-all (car nr) qp): if true, then (car nr) is an acceptable value (no conflicts) for the next queen position. We increment c and cons (car nr) onto the head of qp to add the queen position for the new column. We also update (car nr) to (+ 1 (car nr)) to make a note that the qp position has already been tried, in case we backtrack to this column in the future. We then cons 0 onto the front of nr to signal that we are starting the search for the queen position of a new column.

Keep searching for conflict-free row for current column: (else): if none of the previous conditions are true, then the current value of (car nr) is not a conflict-free choice for next queen position. We must stay on column c but update (car nr) to try the next possible row for the current column. We leave c and qp unchanged, and replace (car nr) with (+ 1 (car nr)).

The **solve-h** function does all the work of controlling the search by means of recursion with the correct parameters updated. The general structure of solve-h is:

```
(define (solve-h n c qp nr sv)
        (cond
                (( end of search )                sv)
                (( found one solution )           (solve-h …))
                (( found row for cur col )        (solve-h …))
                (( try another row for cur col )  (solve-h …))
                (( backtrack to prev col )        (solve-h …))
        )
)
```

So search proceeds by many recursive calls to **(solve-h …).** Parameters must be updated carefully based on the current case within the conditional. When the end of search condition is found, the recursion ends and final solution is sv, the list of lists of all solutions found during the search. Once you've combined the definitions for the solve and nc functions, you can run (solve n) to find all solutions for a problem of size n:

```
> (solve 8)                          solve the problem for 8-by-8 board
'((4 6 1 5 2 0 3 7)                  result is sv, a list of list of all 92 qp solutions found
  (3 6 4 1 5 0 2 7)
  (5 3 6 0 2 4 1 7)
  (5 2 4 6 0 3 1 7)
…
  (4 1 3 6 2 7 5 0)
  (3 1 6 2 5 7 4 0))
>
```

Trace

The following takes an extract from a complete trace for finding all 92 solutions for n=8. It is abbreviated here to provide some commentary for a few useful steps. A (very long) detailed trace for finding all solutions will be posted for reference in a separate document.

> (solve 8)

trace: c: 1 qp: (0) nr: (0 1)

Solution starts with by initializing

qp: ‘(0)

which means column 0 set to row 0, and

nr: '(0 1)

which means column 0 set to row 1 and column 1 set to 0.

trace: c: 1 qp: (0) nr: (1 1)

trace: c: 1 qp: (0) nr: (2 1)

trace: c: 2 qp: (2 0) nr: (0 3 1)

After several conflict checks, row 2 for col 1 is found to have no conflicts with column 0, and is added to qp to become the queen position for column 1. nr is updated to show that 3 is the next row to be tried for column 1, and 0 is added for the new column 3.

trace: c: 2 qp: (2 0) nr: (1 3 1)

trace: c: 2 qp: (2 0) nr: (2 3 1)

trace: c: 2 qp: (2 0) nr: (3 3 1)

trace: c: 2 qp: (2 0) nr: (4 3 1)

trace: c: 3 qp: (4 2 0) nr: (0 5 3 1)

Row 4 for col 2 is found to have no conflicts with cols 0 and 1 of qp and is added to qp col 2. nr is updated with 4+1=5 for col 2, new 0 is added for col 3.

trace: c: 3 qp: (4 2 0) nr: (1 5 3 1)

trace: c: 4 qp: (1 4 2 0) nr: (0 2 5 3 1)

Row 1 for col 3 is found to have no conflicts with cols 0, 1, 2 and is added to qp to become qp for col 3. nr is updated with 1+1=2 for col 3, new 0 added for col 4

trace: c: 4 qp: (1 4 2 0) nr: (1 2 5 3 1) ; (car nr) is 1, but has row conflict with qp (1 4 2 0)

trace: c: 4 qp: (1 4 2 0) nr: (2 2 5 3 1) ; (car nr) is 2, but has row conflict with qp (1 4 2 0)

trace: c: 4 qp: (1 4 2 0) nr: (3 2 5 3 1) ; (car nr) is 3, success, no conflicts with qp (1 4 2 0)

trace: c: 5 qp: (3 1 4 2 0) nr: (0 4 2 5 3 1) ; qp updated with 3 for new col, nr updated (0 4 …)

trace: c: 5 qp: (3 1 4 2 0) nr: (1 4 2 5 3 1)

trace: c: 5 qp: (3 1 4 2 0) nr: (2 4 2 5 3 1)

trace: c: 5 qp: (3 1 4 2 0) nr: (3 4 2 5 3 1)

trace: c: 5 qp: (3 1 4 2 0) nr: (4 4 2 5 3 1)

trace: c: 5 qp: (3 1 4 2 0) nr: (5 4 2 5 3 1)

trace: c: 5 qp: (3 1 4 2 0) nr: (6 4 2 5 3 1)

trace: c: 5 qp: (3 1 4 2 0) nr: (7 4 2 5 3 1)

trace: c: 5 qp: (3 1 4 2 0) nr: (8 4 2 5 3 1) **; (= (car nr) n), so must backtrack to previous col**

trace: c: 4 qp: (1 4 2 0) nr: (4 2 5 3 1)

For column 5 we encounter a problem, since all possible rows 0-7 have a conflict with one of the previous columns 0-4. (car nr) equals 8, meaning all rows for col 5 have been tried. We must

backtrack to the previous column, col 4. We replace qp with (cdr qp) and nr with (cdr nr) and decrement c from 5 to 4. This causes the solution for col 4 to be discarded. The next available row for column 4 is whatever was where we left off earlier which was row 4. We pick up at that point and then try to move forward again.

trace: c: 4 qp: (1 4 2 0) nr: (5 2 5 3 1)
trace: c: 4 qp: (1 4 2 0) nr: (6 2 5 3 1)
trace: c: 4 qp: (1 4 2 0) nr: (7 2 5 3 1) **; no conflicts for row 7 col 5, advance to next column**
trace: c: 5 qp: (7 1 4 2 0) nr: (0 8 2 5 3 1)

We find another row, row 7, for column 4 and can now move forward again to column 5. However we can see that if we ever backtrack to column 4, its next row value is already set to 8, and would trigger another backtrack to row 3. In this way it can happen that one backtrack can result in a cascade of backtracks across multiple columns.

trace: c: 5 qp: (7 1 4 2 0) nr: (1 8 2 5 3 1)
trace: c: 5 qp: (7 1 4 2 0) nr: (2 8 2 5 3 1)
trace: c: 5 qp: (7 1 4 2 0) nr: (3 8 2 5 3 1)
trace: c: 5 qp: (7 1 4 2 0) nr: (4 8 2 5 3 1)
trace: c: 5 qp: (7 1 4 2 0) nr: (5 8 2 5 3 1)
trace: c: 5 qp: (7 1 4 2 0) nr: (6 8 2 5 3 1)
trace: c: 5 qp: (7 1 4 2 0) nr: (7 8 2 5 3 1)
trace: c: 5 qp: (7 1 4 2 0) nr: (8 8 2 5 3 1)
trace: c: 4 qp: (1 4 2 0) nr: (8 2 5 3 1) **; all rows for current col exhausted, backtrack**
trace: c: 3 qp: (4 2 0) nr: (2 5 3 1)

As predicted, we backtrack from column 5 to 4, but then we immediately backtrack again from 4 to 3. Note that for c=5, all 8 rows 0-7 were considered but rejected because of some conflict with a previous column. During a series of steps like these, c and qp remain unchanged, and (car nr) is incremented to move to the next row for column c.

trace: c: 3 qp: (4 2 0) nr: (3 5 3 1)
trace: c: 3 qp: (4 2 0) nr: (4 5 3 1)
trace: c: 3 qp: (4 2 0) nr: (5 5 3 1)
…

Skipping ahead a few lines, let's see what happens when a solution is found, and how we move forward from one solution to begin the search for the next.

…
trace: c: 6 qp: (6 2 5 7 4 0) nr: (1 7 3 6 8 5 1)
trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (0 2 7 3 6 8 5 1)
trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (1 2 7 3 6 8 5 1)
trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (2 2 7 3 6 8 5 1)
trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (3 2 7 3 6 8 5 1)
trace: c: 8 qp: (3 1 6 2 5 7 4 0) nr: (0 4 2 7 3 6 8 5 1) **; (= c 8) is true, so qp is a new solution**
solution 1 (3 1 6 2 5 7 4 0)

Backtrack to col 7, discarding qp for col 7; continue as before to find the next solution

trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (4 2 7 3 6 8 5 1)
trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (5 2 7 3 6 8 5 1)

trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (6 2 7 3 6 8 5 1)
trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (7 2 7 3 6 8 5 1)
trace: c: 7 qp: (1 6 2 5 7 4 0) nr: (8 2 7 3 6 8 5 1)
trace: c: 6 qp: (6 2 5 7 4 0) nr: (2 7 3 6 8 5 1)
trace: c: 6 qp: (6 2 5 7 4 0) nr: (3 7 3 6 8 5 1)
…

Once a conflict-free row value is found for column 7, then we have found a complete solution. The code for advancing to the next column will temporarily add a 9th column with row value 0 to nr. After recording the current solution, we replace qp with (cdr qp) to delete the last value from the current solution to prepare for finding the next one. Also, make two adjustments to nr. We first replace nr with (cdr nr) to discard the 0 entry for the invalid extra column, then we replace (car nr) with (+ 1 (car nr)) to update the next row to try since the original value has already been tried. Begin the search for the next solution, picking up where the previous solution left off.

Skipping ahead again, let's see what happens when we reach the step where the last solution is determined.

…
trace: c: 2 qp: (5 7) nr: (8 6 8)
trace: c: 1 qp: (7) nr: (6 8)
trace: c: 1 qp: (7) nr: (7 8)
trace: c: 1 qp: (7) nr: (8 8)
trace: c: 0 qp: () nr: (8) **; all rows for col 0 exhausted, no col "-1" to backtrack to ### end of search, all solutions found**

As we reach the end of the search, we eventually backtrack all the way to column 0. Once we have tried all rows for column 0, we are at a dead end for col 0. Since the only backtrack option would be to backtrack to column -1, which doesn't exist, then we have our signal that the search has completed. All positions have been checked and all solutions have been found.

Use this trace extract and the complete trace document if you are interested to help you write and correct your implementation of the nc and solve functions.

Testing for n = 8, 9, 10, 11, …

You can check the table of total number of solutions of N Queens for different values of n. Once the list of solutions is found, quickly check the length of the solution vector to confirm the number of solutions found is the same as documented in the table in the Wikipedia article.

(define sv (solve 8))	sv is the list of all solutions for n=8
(length sv)	should evaluate to 92

The time complexity is exponential in n. You should be able to find solutions up to n=11 or 12, but at some point you will run out of stack or the solution execution will take too much time. If you're curious, you can increase the stack space for DrRacket and see if this allows you to find additional solutions.

N Queens in Prolog

The prolog solution will be close to the Racket solution. One important difference is that in Prolog, there is no need to implement any bookkeeping or backtracking since these features are built in. We merely describe the constraints and some choice points, then let Prolog take control of finding solutions by advancing and backtracking as needed.

No Conflict Check Predicate nc/3

nc(V,P,D)

- **V: row index for a new column to be added to existing solution list**
- **P: existing solution list**
- **D: distance (number of columns) between V and head of P, initially D will be 1, but will be incremented during recursion.**

```
nc(_,[],_).          % base case; no conflicts if list is empty, regardless of other arguments

nc(V,P,D) :-
        P = [H|T],
        …            % check for no row and no diagonal conflicts here
        …            % call nc/3 recursively on tail of P and D+1, V unchanged
```

As explained above, **nc/3** only checks that there is no conflict between the new value V and the existing items in list P. It does not check for conflicts among the items already in the list. This extra check would be required for the permutation solution approach, but we don't need to do it here.

Solution Predicate solve/2, solve/3

To find a solution we query the predicate

solve(8,QP).

This predicate will find **one** solution for the N=8 problem. An important adjustment in your thinking about coding the solution is that you don't need to code the bookkeeping, backtracking, and searching for multiple solutions in Prolog, since Prolog does those tasks automatically. What you must do instead is write your solution in a way that works with Prolog backtracking. Specifically you should use predicates such as between/3 which act as a choice point within your predicates. When backtracking is needed, Prolog will backtrack to a choice point to try another value. So you must provide some choice points to help Prolog do its job. Also, we only have to write a predicate to find **one** solution. If we want to get more than one solution or all the solutions, we simply invoke backtracking, either interactively, or in batch mode with the findall/3 predicate.

We will continue to use the one-col-at-a-time approach. The solve/2 predicate is top-level and calls solve/3 helper function with an extra parameter to represent the temporary solution QPT. At the end of the search we match QPT to QP to get our final answer.

```
solve(N,QP) :-
        solve(N,[],QP).
```

The 2nd parameter is the temporary solution which is initially the empty list [ ].

solve/3

```
solve(N,QPT,QP) :-
        …
```

The body of solve/3 takes some steps similar to what we did in the Racket solution. We assume that QPT represents a partial solution that has already been checked for conflicts. We now have to find a row value for a new column and add it to QPT if it passes the no conflict test.

To get a potential row value for a new column, use the following approach:

```
NM1 is N-1,
between(0,NM1,V),
…
```

This goal does a couple of important things. First, it generates a potential row value on the correct range which is 0 to N-1. Second, by using the between/3 predicate, we establish a **choice point** which Prolog can backtrack to if needed.

Now all that remains is to check value V for no conflicts with the temporary solution QPT at distance D = 1. If the nc goal succeeds, then we proceed to add V to QPT, yielding another list QPT1, and calling solve/3 recursively with QPT replaced with QPT1. What if the nc goal fails? No problem, Prolog backtracks to the between/3 choice point and picks another value of V. What if between/3 runs out of values of V? No problem, Prolog backtracks to an earlier recursive call with an earlier call to between/3 and uses that as another choice point, and so on.

The only remaining rule is a rule to stop when the temporary solution QPT is a complete solution. We note that QPT started off as the empty list []. Each recursive step we add another column, so QPT grows by 1 each recursive call. We stop when length of QPT reaches N. When QPT has length N, we can match it with QP and get our final answer. Write an additional rule for solve/3 that matches length of QPT to N and then matches QPT to QP.

Multiple Solutions With Backtracking

We will run the solve/2 predicate as a query to get one solution.

```
?- solve(8,QP).
QP = [3, 1, 6, 2, 5, 7, 4, 0] ;
QP = [4, 1, 3, 6, 2, 7, 5, 0] ;
QP = [2, 4, 1, 7, 5, 3, 6, 0] ;
QP = [2, 5, 3, 1, 7, 4, 6, 0] .
```

Since Prolog knows about the choice points in our predicates, it prompts us for interactive backtracking to get multiple solutions if we want. So enter ":" or "." as desired to see more than one solution. If we want all the solutions at once (batch mode) we can get them via findall/3.

```
?- findall(QP, solve(8,QP), QPALL).
QPALL = [[3, 1, 6, 2, 5, 7, 4, 0], [4, 1, 3, 6, 2, 7, 5|...], [2, 4, 1, 7, 5, 3|...], [2, 5, 3, 1, 7|...], [4, 6, 0,
2|...], [3, 5, 7|...], [2, 5|...], [4|...], [...|...]|...].
```

In this query, QP holds one solution at a time as generated by solve(8,QP). We instruct findall to do batch mode backtracking and collect all individual solutions QP into a list of lists QPALL.

```
?- findall(QP, solve(8,QP), QPALL), length(QPALL,L).
QPALL = [[3, 1, 6, 2, 5, 7, 4, 0], [4, 1, 3, 6, 2, 7, 5|...], [2, 4, 1, 7, 5, 3|...], [2, 5, 3, 1, 7|...], [4, 6, 0,
2|...], [3, 5, 7|...], [2, 5|...], [4|...], [...|...]|...],
L = 92.
```

Here, we find all solutions in QPALL, and without looking at them individually, we confirm that it contains the correct number 92.

```
?- findall(QP, solve(8,QP), QPALL), forall(member(QP1,QPALL),writeln(QP1)).
[3,1,6,2,5,7,4,0]
[4,1,3,6,2,7,5,0]
[2,4,1,7,5,3,6,0]
[2,5,3,1,7,4,6,0]
[4,6,0,2,7,5,3,1]
…
[5,3,6,0,2,4,1,7]
[3,6,4,1,5,0,2,7]
[4,6,1,5,2,0,3,7]
QPALL = [[3, 1, 6, 2, 5, 7, 4, 0], [4, 1, 3, 6, 2, 7, 5|...], [2, 4, 1, 7, 5, 3|...], [2, 5, 3, 1, 7|...], [4, 6, 0,
2|...], [3, 5, 7|...], [2, 5|...], [4|...], [...|...]|...].
```

Here, after finding all the solutions in QPALL, we use forall/2 and member/2 to select each solution from QPALL and write them to the display. member(QP1,QPALL) sets QP1 to one of the solutions in QPALL which we then display with writeln(QP1). The forall/2 predicate then runs member/2 for each QP1 that is a member of QPALL.

Graphical Display of Solution (Optional)

By using between/3, forall/2, and write/writeln, you can display a simple graphic of the chess board with the queen positions displayed.

```
?- solve(8,QP),display_solution(QP).

  0  1  2  3  4  5  6  7
 ------------------------
0|  |  |  |  |  |  |  | X|
 ------------------------
1|  | X|  |  |  |  |  |  |
 ------------------------
2|  |  |  | X|  |  |  |  |
 ------------------------
3| X|  |  |  |  |  |  |  |
 ------------------------
4|  |  |  |  |  |  | X|  |
 ------------------------
5|  |  |  |  | X|  |  |  |
 ------------------------
6|  |  | X|  |  |  |  |  |
 ------------------------
7|  |  |  |  |  | X|  |  |
 ------------------------

QP = [3, 1, 6, 2, 5, 7, 4, 0]
```

Choice of font is important when doing an ASCII drawing, so pick a monospace font like Lucida Console to get all the columns to line up.

Predicates to Create Display Graphic

display_solution(QP)

This predicate should check the length of QP to get N, then for each row R call display_row(R,QP). Use the following approach

forall(between(0,NM1,R), (some task involving R goes here))

Use write(…) to write a string or write a value of a variable. Use writeln(“”) to write a newline.

display_row(R,QP)

Check the length of QP and display each column in the row

forall(between(0,NM1,C), (some task involving R and C goes here))

The display_row predicate should call display_square for each column. This predicate is not recursive or iterative, it either displays a space or an X to mark a queen position if there is a queen positioned in that column. For example, here is a possible implementation of display_square:

```
display_square(R,C,QP) :-
   nth0(C,QP,V),                    % get value V from index C in list QP
   (   R =:= V                      % if V equals the current row value R
   ->  write(" X|")                 % then it is a queen position, display “X”
   ;   write("  |")                 % else that column is empty, display space
   ).
```

You’ll also need some helper predicates to display a row of column headers and a predicate to display a line of symbols for a row separator. Use the forall/between approach described above to do repetitive display. These can be called from display_solution and display_row as you wish.

display_col_headers(N)

display_row_separator(N)