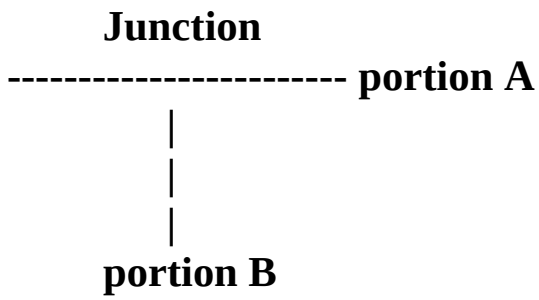
Backtracking

Backtracking is a technique used to solve problems with a large search space, that systematically tries and eliminates possibilities.

A standard example of backtracking would be going through a maze. At some point in a maze, you might have two options of which direction to go:



One strategy would be to try going through portion A of the maze. If you get stuck before you find your way out, then you *"backtrack"* to the junction. At this point in time you know that portion A will NOT lead you out of the maze, so you then start searching in portion B.

Clearly, at a single junction you could have even more than two choices. The backtracking strategy says to try each choice, one after the other, if you ever get stuck, *"backtrack"* to the junction and try the next choice. If you try all choices and never found a way out, then there IS no solution to the maze.

Eight Queens Problem

The problem is specified as follows:

Find an arrangement of eight queens on a single chess board such that no two queens are attacking one another.

In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).

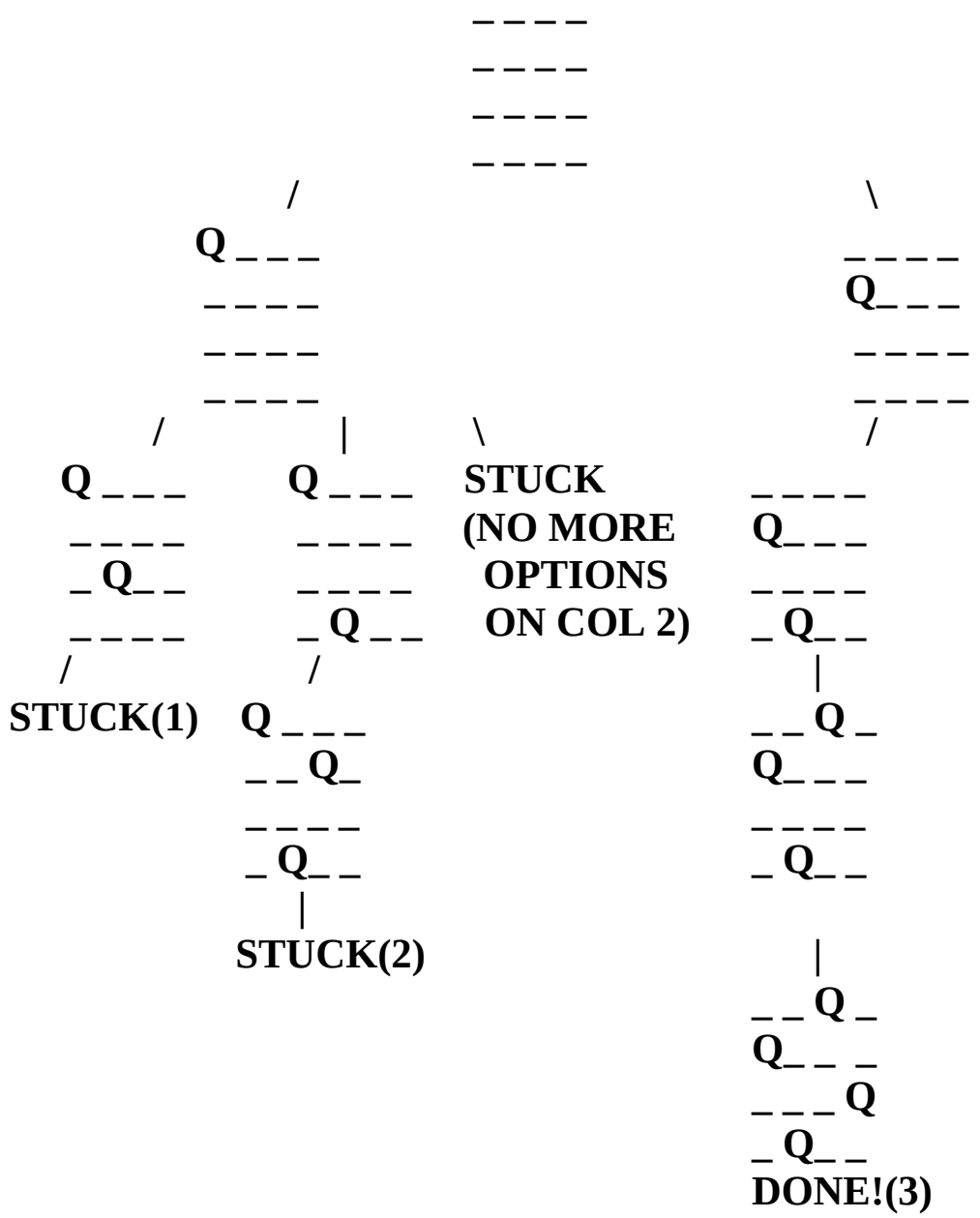
Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.

The backtracking strategy is as follows:

1) Place a queen on the first available square in row 1.
2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
3) Continue in this fashion until either (a) you have solved the problem, or (b) you get stuck. When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.

When we carry out backtracking, an easy way to visualize what is going on is a tree that shows all the different possibilities that have been tried.

Consider the following page with a visual representation of solving the 4 Queens problem (placing 4 queens on a 4x4 board where no two attack one another).


STUCK(1)
STUCK(2)
STUCK
(NO MORE
OPTIONS
ON COL 2)
DONE!(3)

The neat thing about coding up backtracking, is that it can be done recursively, without having to do all the bookkeeping at once.

Instead, the stack or recursive calls does most of the bookkeeping (ie, keeping track of which queens we've placed, and which combinations we've tried so far, etc.)

Here is some code that is at the heart of the eight queens solution:

```
void solveItRec(int perm[], int location, struct onesquare usedList[]) {

  int i;

  if (location == SIZE) {
     printSol(perm);
  }

  for (i=0; i<SIZE; i++) {

     if (usedList[i].selected == 0) {

        if (!conflict(perm, location, usedList[i].row)) {

            perm[location] = usedList[i].row;
            usedList[i].selected = 1;
            solveItRec(perm, location+1, usedList);
            usedList[i].selected = 0;
         }
      }
   }
}
```

Sudoku and Backtracking

Another common puzzle that can be solved by backtracking is a Sudoku puzzle. The basic idea behind the solution is as follows:

1) Scan the board to look for an empty square that could take on the fewest possible values based on the simple game constraints.
2) If you find a square that can only be one possible value, fill it in with that one value and continue the algorithm.
3) If no such square exists, place one of the possible numbers for that square in the number and repeat the process.
4) If you ever get stuck, erase the last number placed and see if there are other possible choices for that slot and try those next.

Mazes and Backtracking

A final example of something that can be solved using backtracking is a maze. From your start point, you will iterate through each possible starting move. From there, you recursively move forward. If you ever get stuck, the recursion takes you back to where you were, and you try the next possible move.

In dealing with a maze, to make sure you don't try too many possibilities, one should mark which locations in the maze have been visited already so that no location in the maze gets visited twice. (If a place has already been visited, there is no point in trying to reach the end of the maze from there again.

Tentaizu

In this problem, you are given a 7 x 7 board where there are exactly 10 hidden bombs. On the board, some squares have numbers on them, representing the number of adjacent bombs to that square, just like the numbers in the game, Minesweeper. The numbers are given in such a way that there is only one arrangements of 10 bombs that is consistent with the numbers given. The goal of the problem is to determine where all of the 10 bombs go.

Regular brute force would try up to $\binom{49}{10}$ combinations of placements of stars. This is far too many to run in a reasonable amount of time.

But…this is where backtracking comes in. Consider placing bombs in squares one by one, from top to bottom, going left to right in each row. A vast majority of possible combinations can be eliminated very early on due to inconsistencies!!!

Thus, our idea is as follows:

1) **Step through the board, in row major order (row by row from top to bottom, and from left to right in each row).**
2) **For each square, try placing a bomb in it and recursively see if this solution works.**
3) **If it doesn't, then try skipping the square and see if that solution works.**

Thus, our recursive function needs two critical pieces of information:

1) **Which square I am considering**
2) **The number of bombs already placed.**

The key to back-tracking is effectively cutting off "doomed" partial solutions as early as possible. In our recursion, this corresponds to base cases in the beginning – typically if statements to handle "easy" cases.

For our problem, it's very easy to test consistency issues just with the number of bombs already placed and our current position.

Then, we can test more detailed consistency issues.

Finally, we move onto our recursive case(s).

For some squares, we can't place a bomb (these are squares that already have a number or that are adjacent to a 0). For these squares, we make one recursive call only, corresponding to not placing a bomb in that squre.

For other squares we try two things:

1) **Place the bomb and see what the recursion returns.**
2) **If that didn't work, undo that bomb and run the recursion again.**

In the code, the recursive function solveRec is structured very similar to the description given above.