Examples of Recursive Backracking

Today we will begin to examine problems with several possible recursive decompositions from one instance of the problem to another. That is, each time we make a recursive call, we will have to make a **choice** as to which step to take. Potentially we may want to try all possible steps to find the optimal one, or exhaustively try out all possibilities. But sometimes, we are only interested in one solution, whichever one that works out first. At each decision point, we will make a choice from one of the available options, and we choose the wrong one, we will eventually run into a dead end and find ourselves in a state from which we are unable to solve the problem immediately and unable to go further; when this happens, we will have to **backtrack** to a "decision point" and try another alternative.

If we ever solve the problem, great, we're done. Otherwise, we need to keep exploring all possible paths by making choices and, when they prove to have been wrong, backtracking to the most recent choice point. What's really interesting about backtracking is that we only back up in the recursion as far as we need to go to reach a previously unexplored decision point. Eventually, more and more of these decision points will have been explored, and we will backtrack further and further. If we happen to backtrack to our initial position and find ourselves with no more choices from that initial position, the particular problem at hand is unsolvable.

As was the case with recursion, simply discussing the idea doesn't usually make the concepts transparent, it is therefore worthwhile to look at many examples until the idea of backtracking and the approach taken to solve problems begins to make sense. This handout contains code for generating permutations, the queen attack problem, and solving cryptarithmetic puzzles. As before, the code can be complex and is somewhat sparsely commented, you should make sure to keep up with the discussion in lecture. The maze backtracking example is fully covered in your textbook as an alternate example to study.

Permutations

First, another procedural recursion example, this one from the text that forms all possible re-arrangements of the letters in a string. It is an example of an exhaustive procedural algorithm. The pseudocode strategy is as follows:

> *If you have no more characters left to rearrange, print current string*
> *for (every possible choice among the characters left to rearrange) {*
> *Make that choice and swap that character to the front of the string*
> *Use recursion to rearrange the letters after the new front most*
> *Back out of that choice and restore the state at the beginning of the loop*
> *}*

Here is the code at the heart of that algorithm:

```
static void RecursivePermute(string s, int k)
{
   int i;

   if (k == StringLength(s))
      printf("%s\n", s);
   } else {
```

```
      for (i = k; i < StringLength(s); i++) {
         Swap(&s[k], &s[i]);
         RecursivePermute(s, k+1);
         Swap(&s[k], &s[i]);
      }
   }
}
```

In this exhaustive traversal, we try every possible combination. There are n! ways to rearrange the characters in a string of length n and this prints all of them. In the bactracking examples to follow, we will try one and only reverse that decision if it doesn't work out.

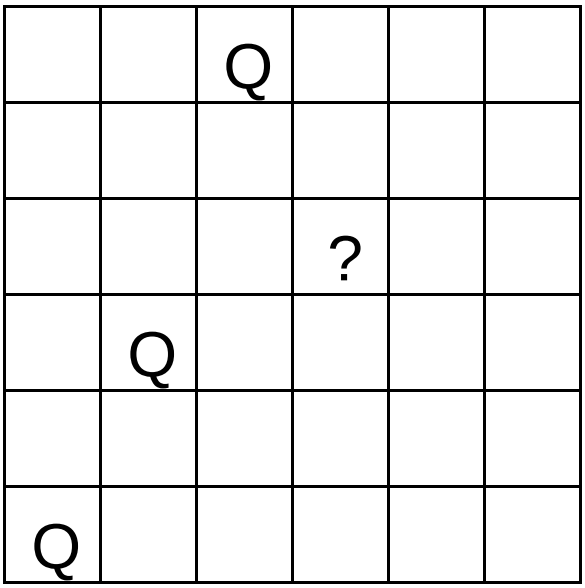
The 8-Queens Problem

The goal here is to assign eight queens to eight positions on a 8x8 chessboard so that no queen, according to the rules of normal chess play, can attack any other queen on the board.

In pseudocode, our strategy will be:

Start in the leftmost columm

If all queens are placed, return TRUE
for (every possible choice among the rows in this column)
if the queen can be placed safely there,
make that choice and then recursively try to place the rest of the queens
if recursion sucessful, return TRUE
if !successful, remove queen and try another row in this column
if all rows have been tried and nothing worked, return FALSE to trigger backtracking




```
/*
 * File: queens.c
 * --------------
 * This program implements a graphical search for a solution to the N
 * N queens problem, utilizing a recursive backtracking approach. See
 * comments for Solve function for more details on the algorithm.
 */
```

```
#include "chessGraphics.h"     // routines to draw chessboard & queens

#define NUM_QUEENS 8

static bool Solve(bool board[][NUM_QUEENS], int col);
static void PlaceQueen(bool board[][NUM_QUEENS], int row, int col);
static void RemoveQueen(bool board[][NUM_QUEENS], int row, int col);
static bool LeftIsClear(bool board[][NUM_QUEENS], int row, int col);
static bool UpperDiagIsClear(bool board[][NUM_QUEENS], int row, int col);
static bool LowerDiagIsClear(bool board[][NUM_QUEENS], int row, int col);
static bool IsSafe(bool board[][NUM_QUEENS], int row, int col);
static void ClearBoard(bool board[][NUM_QUEENS]);

main()
{
       bool board[NUM_QUEENS][NUM_QUEENS];

       InitGraphics();
       ClearBoard(board);            // Set all board positions to false
       DrawChessboard(NUM_QUEENS);   // Draw empty chessboard of right size
       Solve(board,0);               // Attempts to solve the puzzle
}

/*
 * Function: Solve
 * ---------------
 * This function is the main entry in solving the N queens problem. It takes
 * the partially filled board and the column we are trying to place queen in.
 * It will return a boolean value which indicates whether or not we found a
 * successful arrangement starting from this configuration.
 *
 * Base case:  if there are no more queens to place, we have succeeded!
 *
 * Otherwise, we find a safe row in this column, place a queen at (row,col)
 * recursively call Solve starting at the next column using this new board
 * configuration. If that Solve call fails, we remove that queen from (row,col)
 * and try again with the next safe row within the column.  If we have tried all
 * rows in this column and haven't found a solution, we return FALSE from this
 * invocation, which will force backtracking from this unsolvable configuration.
 *
 * The starting call to Solve has an empty board and is placing a queen in col 0:
 *        Solve(board, 0);
 */
static bool Solve(bool board[][NUM_QUEENS], int col)
{
    int rowToTry;

    if (col >= NUM_QUEENS) return TRUE;                // base case

    for (rowToTry = 0; rowToTry < NUM_QUEENS; rowToTry++)  {
        if (IsSafe(board,rowToTry,col)) {
           PlaceQueen(board, rowToTry, col);           // try queen
           if (Solve(board, col + 1)) return TRUE;     // recursively try rest
           RemoveQueen(board, rowToTry, col);          // failed, try again
        }
    }
    return FALSE;           // no safe place for queen given this configuration
```

```
/*
 * Function: PlaceQueen
 * --------------------
 * Places a queen in (row,col) of the board by setting value in
 * array to TRUE and drawing a 'Q' in that square on the displayed chessboard.
 */
static void PlaceQueen(bool board[][NUM_QUEENS], int row, int col)
{
       board[row][col] = TRUE;
       DrawLetterAtPosition('Q', row, col);  // Function lifted from Boggle
}


/*
 * Function: Remove Queen
 * ----------------------
 * Removes a queen from (row,col) of the board by setting value in array to
 * FALSE and erasing the 'Q' from that square on the displayed chessboard.
 */
static void RemoveQueen(bool board[][NUM_QUEENS], int row, int col)
{
       board[row][col] = FALSE;
       EraseLetterAtPosition('Q', row, col);
}


/*
 * Function: IsSafe
 * ----------------
 * Given a partially filled board and (row,col), returns boolean value
 * which indicates whether that position is safe (i.e. not threatened by
 * another queen already on the board.)
 */
static bool IsSafe(bool board[][NUM_QUEENS], int row, int col)
{
    return (LeftIsClear(board, row, col) &&
            UpperDiagIsClear(board, row, col) &&
            LowerDiagIsClear(board, row, col));
}

/*
 * Function: LeftIsClear
 * ---------------------
 * Given a partially filled board and (row,col), checks that there
 * is no queen in this row in any column to the left.
 */
static bool LeftIsClear(bool board[][NUM_QUEENS], int row, int col)
{
       int colToCheck;

       for (colToCheck = 0; colToCheck < col; colToCheck++)
          if (board[row][colToCheck]) return FALSE;
    return TRUE;
}
```

```
/*
 * Function: UpperDiagIsClear
 * --------------------------
 * Given a partially filled board and (row,col), checks that there
 * is no queen along the northwest diagonal that crosses (row,col).
 */
static bool UpperDiagIsClear(bool board[][NUM_QUEENS], int row, int col)
{
       int x, y;

       for (x = col, y = row; x >= 0 && y < NUM_QUEENS; x--, y++)
       if (board[y][x]) return FALSE;
       return TRUE;
}

/*
 * Function: LowerDiagIsClear
 * --------------------------
 * Given a partially filled board and (row,col), checks that there
 * is no queen along the southwest diagonal that crosses (row,col).
 */
static bool LowerDiagIsClear(bool board[][NUM_QUEENS], int row, int col)
{
       int x, y;

       for (x = col, y = row; x >= 0 && y >=0; x--, y--)
          if (board[y][x]) return FALSE;
    return TRUE;
}

/*
 * Function: ClearBoard
 * --------------------
 * Simply initializes the board to be empty, i.e. no queens is on any square.
 */
static void ClearBoard(bool board[][NUM_QUEENS])
{
       int row, col;

       for (row = 0; row < NUM_QUEENS; row++)
       for (col = 0; col < NUM_QUEENS; col++)
          board[row][col] = FALSE;
}
```

Solving cryptarithmetic puzzles

Newspapers and magazines often have cryptarithmetic puzzles of the form:

```
   SEND
+ MORE
 MONEY
```

The goal here is to assign each letter a digit from 0 to 9 so that the arithmetic works out correctly. The rules are that all occurrences of a letter must be assigned the same digit, and no digit can be assigned to more than one letter. First, I will show you a workable, but not very efficient strategy and then improve on it.

In pseudocode, our first strategy will be:

First, create a list of all the characters that need assigning to pass to Solve

If all characters are assigned, return TRUE if puzzle is solved, FALSE otherwise
Otherwise, consider the first unassigned character
for (every possible choice among the digits not in use)
    make that choice and then recursively try to assign the rest of the characters
    if recursion sucessful, return TRUE
    if !successful, unmake assignment and try another digit
if all digits have been tried and nothing worked, return FALSE to trigger backtracking

And here is the code at the heart of the recursive program (the other parts have been excluded for clarity):

```
/* ExhaustiveSolve
 * ---------------
 * This is the "not-very-smart" version of cryptarithmetic solver. It takes
 * the puzzle itself (with the 3 strings for the two addends and sum) and a
 * string of letters as yet unassigned. If there are no more letters to assign
 * then we've hit a base-case, if the current letter-to-digit mapping solves
 * the puzzle, we're done, otherwise we return FALSE to trigger backtracking
 * If we have letters to assign, we take the first letter from that list, and
 * try assigning it the digits from 0 to 9 and then recursively working
 * through solving puzzle from here. If we manage to make a good assignment
 * that works, we've succeeded, else we need to unassign that choice and try
 * another digit. This version is easy to write, since it uses a simple
 * approach (quite similar to permutations if you think about it) but it is
 * not so smart because it doesn't take into account the structure of the
 * puzzle constraints (for example, once the two digits for the addends have
 * been assigned, there is no reason to try anything other than the correct
 * correct digit for the sum) yet it tries a lot of useless combos regardless.
 */
static bool ExhaustiveSolve(puzzleT puzzle, string lettersToAssign)
{
   int digit;

   if (StringLength(lettersToAssign) == 0) // no more choices to make
       return PuzzleSolved(puzzle); // checks arithmetic to see if works
   for (digit = 0; digit <= 9; digit++) { // try all digits
       if (AssignLetterToDigit(lettersToAssign[0], digit)) {
           if (ExhaustiveSolve(puzzle, lettersToAssign + 1))
               return TRUE;
           UnassignLetterFromDigit(lettersToAssign[0], digit);
       }
   }
   return FALSE;  // nothing worked, need to backtrack
}
```

The algorithm above actually has a lot in common with the permuations algorithm, it pretty much just creates all arrangements of the mapping from characters to digits and tries each until one works or all have been unsuccessfully tried. For a large puzzle, this could take a while.

A smarter algorithm could take into account the structure of the puzzle and avoid going down dead-end paths. For example, if we assign the characters starting from the ones place and moving

to the left, at each stage, we can verify the correctness of what we have so far before we continue onwards. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

Our psuedocode in this case has more special cases, but the same general design:

Start by examining the rightmost digit of the topmost row, with a carry of 0

If we are beyond the leftmost digit of the puzzle, return TRUE if no carry, FALSE otherwise
If we are currently trying to assign a char in one of the addends
If char already assigned, just recur on row beneath this one, adding value into sum
If not assigned, then
for (every possible choice among the digits not in use)
make that choice and then on row beneath this one, if successful, return TRUE
if !successful, unmake assignment and try another digit
return FALSE if no assignment worked to trigger backtracking
Else if trying to assign a char in the sum
If char assigned & matches correct, recur on next column to the left with carry, if success return TRUE
If char assigned & doesn't match, return FALSE
If char unassigned & correct digit already used, return FALSE
If char unassigned & correct digit unused, assign it and recur on next column to left with carry, if success return TRUE
return FALSE to trigger backtracking

```
/* SmarterSolve
 * ------------
 * This is the more clever version of cryptarithmetic solver.
 * It takes the puzzle itself (with the 3 strings for the two addends
 * and the sum), the row we are currently trying to assign and the
 * sum so far for that column. We are always assumed to be working on
 * the last column of the puzzle (when we finish a column, we remove
 * that column and solve the smaller version of the puzzle consisting
 * of the remaining columns).  (One special thing to note is that
 * the strings have been reversed to make the processing slightly
 * easier for this version). The base case for this version is
 * when we have assigned all columns and thus the strings that
 * remain are empty. As long as we didn't come in with any carry,
 * we must have successfully assigned the letters.
 * If we aren't at a base case, the two recursive steps are
 * separated into helper functions, one for assigning an addend,
 * the other for handling the sum.
 * This version is more complex to write, since it tries to take
 * into account the nature of addition to avoid trying a lot
 * of useless combinations unnecesssarily.
 */
```

```
static bool SmarterSolve(puzzleT puzzle, int whichRow, int rowSumSoFar)
{
     if (StringLength(puzzle.rows[Sum]) == 0) /* we've assigned all columns */
         return (rowSumSoFar == 0);      /* if no carry in, we solved it! */
     if (whichRow == Addend1 || whichRow == Addend2)
         return SolveForAddend(puzzle, whichRow, rowSumSoFar);
     else
         return SolveForSum(puzzle, rowSumSoFar);
}


/* SolveForAddend
 * ------------
 * Helper for SmarterSolver to assign one digit in an addend.
 * If there are no letters remaining in that addend or the
 * last letter is already assigned, we just recur on the rest
 * of the puzzle from here. If the letter isn't assigned, we
 * try all the possibilities one by one until we are able to
 * get one that works, or we return FALSE to trigger backtracking
 * on our earlier decisions.
 */
static bool SolveForAddend(puzzleT puzzle, int whichRow, int rowSumSoFar)
{
   char ch;
   int digit;

   if (StringLength(puzzle.rows[whichRow]) == 0)    // addend ended already
   return SmarterSolve(puzzle, whichRow+1, rowSumSoFar);

   ch = puzzle.rows[whichRow][0];
   puzzle.rows[whichRow]++;
   if (LetterUsed(ch))  // already assigned, just go from here
      return SmarterSolve(puzzle, whichRow+1, LetterToDigit(ch) +
rowSumSoFar);
   else  {                    // not yet assigned, try all digits
      for (digit = 0; digit <= 9; digit++) {
         if (AssignLetterToDigit(ch, digit)) {
            if (SmarterSolve(puzzle, whichRow+1, rowSumSoFar + digit))
                return TRUE;
            UnassignLetterFromDigit(ch, digit);
         }
      }
      return FALSE;
   }
}



/* SolveForSum
 * ------------
 * Helper for SmarterSolver to assign one digit in the sum.
 * If the letter is already assigned and it matches the required
 * sum as computed from the digits in the addends, we're fine
 * and just recur on the rest of the puzzle. If the letter is
 * assigned and it doesn't match, we fail and trigger backtracking.
 * If the letter isn't assigned, we assign it the necessary
 * digit and recur.
 */
```

```
static bool SolveForSum(puzzleT puzzle, int rowSum)
{
   char ch = puzzle.rows[Sum][0];
   int sumDigit = (rowSum % 10);

   puzzle.rows[Sum]++;
   if (LetterUsed(ch)) {            // already assigned
      return (LetterToDigit(ch) == sumDigit)
            && SmarterSolve(puzzle, Addend1, rowSum/10);
   } else {
      if (AssignLetterToDigit(ch, sumDigit)) { // only try matching digit
         if (SmarterSolve(puzzle, Addend1, rowSum/10))
            return TRUE;
         UnassignLetterFromDigit(ch, sumDigit);
      }
      return FALSE;
   }
}
```