CS106X
Autumn 2009

Handout 14
October 9th, 2009

Recursive Backtracking

Recursive Backtracking

So far, all of the recursive algorithms we have seen have shared one very important property: each time a problem was recursively decomposed into a simpler instance of the same problem, only **one** such decomposition was possible; consequently, the algorithm was guaranteed to produce a solution to the original problem. Today we will begin to examine problems with several possible 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 decomposition to use. If 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 decompose the problem any further; when this happens, we will have to **backtrack** to a "choice point" and try another alternative. You'll want to read Chapter 7 of the reader very carefully. We'll complement the reader with a good number of additional examples not in the reader.

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 choice point. Eventually, more and more of these choice 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.

Finish up Chapters 5 and 6, and start reading Chapter 7. Your Boggle assignment assumes you're familiar with all of the material in Chapters 5 – 7, so make sure you read through it before tackling it.

Shrinking A Word [courtesy of Julie Zelenski]

Consider the simply stated question: Is it possible to take an English word and remove all of its letters in some order such that every string along the way is also an English word?

Sometimes it's possible. For example, we can shrink the word **"smart"** down to the empty string while obeying the restriction that all of the intervening strings are also legitimate words. Check this out:

```
"smart"
"mart"
"art"
"at"
"a"
""
```

We elect to first remove the **s**, then the **m**, then the **r**, then the **t**, and finally the **a**. Note that every single string in the above diagram is an English word. That means that, for the purposes of the this problem, it's possible to shrink the word "**smart**" down to the empty string.

Not surprisingly, there are some perfectly good English words that can't be shrunk at all: **"zephyr"**, **"lymph"**, **"rope"**, **"father"**. A reasonable question to ask at this point: Can we programmatically figure out which words can be shrunk and which ones can't?

The answer, not surprisingly, is yes, and our answer involves a form of recursion we've not formally seen with any of the previous recursion examples.

All prior recursive examples have been fully exhaustive in that every single recursive call that could be made was indeed made and contributed to the overall answer. In this example, we only commit to some of the recursive calls if previous ones failed to provide an answer. Check out the code for our **CanShrink** function, which returns **true** if and only if it's possible to shrink the provided word down to the empty string:

```
bool CanShrink(string str, Lexicon& lex)
{
   if (str.empty()) return true;
   if (!lex.containsWord(str)) return false;

   for (int i = 0; i < str.size(); i++) {
      string subsequence = str.substr(0, i) + str.substr(i + 1);
      if (CanShrink(subsequence, lex)) {
         return true;
      }
   }

   return false;
}
```

The code includes two base bases—that part isn't new. But look at the **`for`** loop and how it surrounds the recursive call. Each iteration considers the incoming string with some character removed, and then looks to see if that new string is also a word and can be shrunk down to nothingness. Look at this **`for`** loop as a series of opportunities to return **`true`**. If it just so happens to find some path to the empty string sooner than later, it returns **`true`** without making any other **`CanShrink`** calls. Only if every single path turns up nothing do we truly give up and return **`false`**.

It's pretty clear why **`true`** and **`false`** get returned, since the initial call wants a yes or no as to whether or not the provided word can be collapsed down to the empty string. But what may not be immediately clear is that the recursive calls also rely on those return values to figure out whether or not the path it chose to recursively explore was a good one.

The above version correctly returns **`true`** or **`false`**, but it doesn't remember what words lead down to the empty string. This second version uses a **`Stack<string>`** to take a snapshot of all of the strings that led down the empty string, and it builds this **`Stack<string>`** as the cascade of return **`true`** statements prompt the recursion to unwind. Here's the new and improved version:

```
bool CanShrink(string str, Lexicon& lex, Stack<string>& path)
{
   if (str.empty()) {
      path.push("");
      return true;
   }

   if (!lex.containsWord(str)) return false;
   for (int i = 0; i < str.size(); i++) {
      string subsequence = str.substr(0, i) + str.substr(i + 1);
      if (CanShrink(subsequence, lex, path)) {
         path.push(subsequence);
         return true;
      }
   }

   return false;
}
```

If the initial call to **`CanShrink`** returns **`true`**, then we know that the **`Stack<string>`** contains the bottom-up accumulation of all of the strings that led to the empty string.

```
void TestShrink()
{
   Lexicon lex("lexicon.dat");
   while (true) {
      cout << "Enter a word: ";
      string word = GetLine();
      if (word.empty()) break;
      Stack<string> path;
      if (CanShrink(word, lex, path)) {
         cout << "Yep, and here's the path:" << endl;
         while (!path.isEmpty()) {
            cout << "\t\"" << path.pop() << "\"" << endl;
         }
      } else {
         cout << "That word can't be shrunk down." << endl;
      }
   }
}
```

Periodic Table as Alphabet: Take II

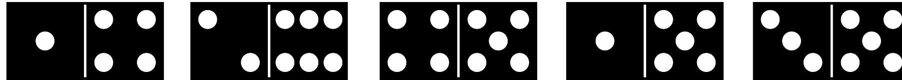
In a previous example, we wrote code that lists all of those English words that can be spelled out using the symbols of the periodic table and nothing else. Here's a related problem that asks specifically whether or not the provided word can be spelled out using just the periodic table symbols. We'll assume that all of the symbols are 1, 2, or 3 letters long, that the symbols come via a **`Set<string>`**, and that all of the **`Lexicon`** methods we're familiar with are case-insensitive.

The idea is to maintain a working prefix of everything so far that's been covered by one or more chemical symbols, and to see if the rest of the word can be completed by figuring out if the next 1, 2, or 3 characters in the string just happen to be a chemical symbol, and if so, recurring on the rest of it. Here's the solution I came up with:

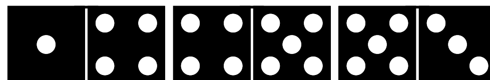
```
bool CanFormWord(string word, Set<string>& symbols)
{
   if (word == "") return true;

   int length = word.size();
   for (int i = 1; i <= min(3, length); i++) {
      if (symbols.contains(word.substr(0, i)) &&
          CanFormWord(word.substr(i), symbols)) {
         return true;
      }
   }

   return false;
}
```

Playing Dominoes [courtesy of Eric Roberts]

The game of dominoes is played with rectangular pieces composed of two connected squares, each of which is marked with a certain number of dots. For example, each of the following five rectangles represents a domino:

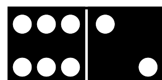


Dominoes are connected end-to-end to form chains, subject to the condition that two dominoes can be linked together only if the numbers match, although it is legal to rotate dominoes 180˚ so that the numbers are reversed. For example, you could connect the first, third, and fifth dominoes in the above collection to form the following chain:



Note that the 3-5 domino had to be rotated so that it matched up correctly with the 4-5 domino.

Given a set of dominoes, an interesting question to ask is whether it is possible to form a chain starting at one number and ending with another. For example, the example chain shown earlier makes it clear that you can use the original set of five dominoes to build a chain starting with a 1 and ending with a 3. Similarly, if you wanted to build a chain starting with a 6 and ending with a 2, you could do so using only one domino:



On the other hand, there is no way—using just these five dominoes—to build a chain starting with a 1 and ending with a 6.

Dominoes can, of course, be represented in C++ very easily as a pair of integers. Assuming the type **`domino`** is defined as

```
struct domino {
   int first;
   int second;
};
```

write a predicate function:

```
bool ChainExists(Vector<domino>& dominoes, int start, int end);
```

that returns **true** if it possible to build a chain from **start** to **finish** using any subset of the **n** dominoes in the **dominoes** simplify. To simplify the problem, assume that **ChainExists** always returns **true** if **start** is equal to **finish** because you can connect any number to itself with a chain of zero dominoes.

For example, if **dominoes** is the domino set illustrated, calling **ChainExists** would give the following results:

```
ChainExists(dominoes, 1, 3)  →  true
ChainExists(dominoes, 5, 5)  →  true
ChainExists(dominoes, 1, 6)  →  false
```

Code

```
bool ChainExists(Vector<domino>& dominoes, int start, int end)
{
   if (start == end) return true;

   for (int i = 0; i < dominoes.size(); i++) {
      domino d = dominoes[i];
      dominoes.removeAt(i);
      if (d.first == start &&
          ChainExists(dominoes, d.second, end)) return true;
      if (d.second == start &&
          ChainExists(dominoes, d.first, end)) return true;
      dominoes.insertAt(i, d); // pretend we never made this choice
   }

   return false;
}
```

Finding Meaningful Subsequences

A subsequence of a string is an ordered subset of its characters. It's like a substring, except that the characters of the subsequence need not be neighbors in the original string. For example, **`"ossia"`** is a subsequence of **`"dorissullivan"`**, because I can cross out all but those five letters and retain what's left:

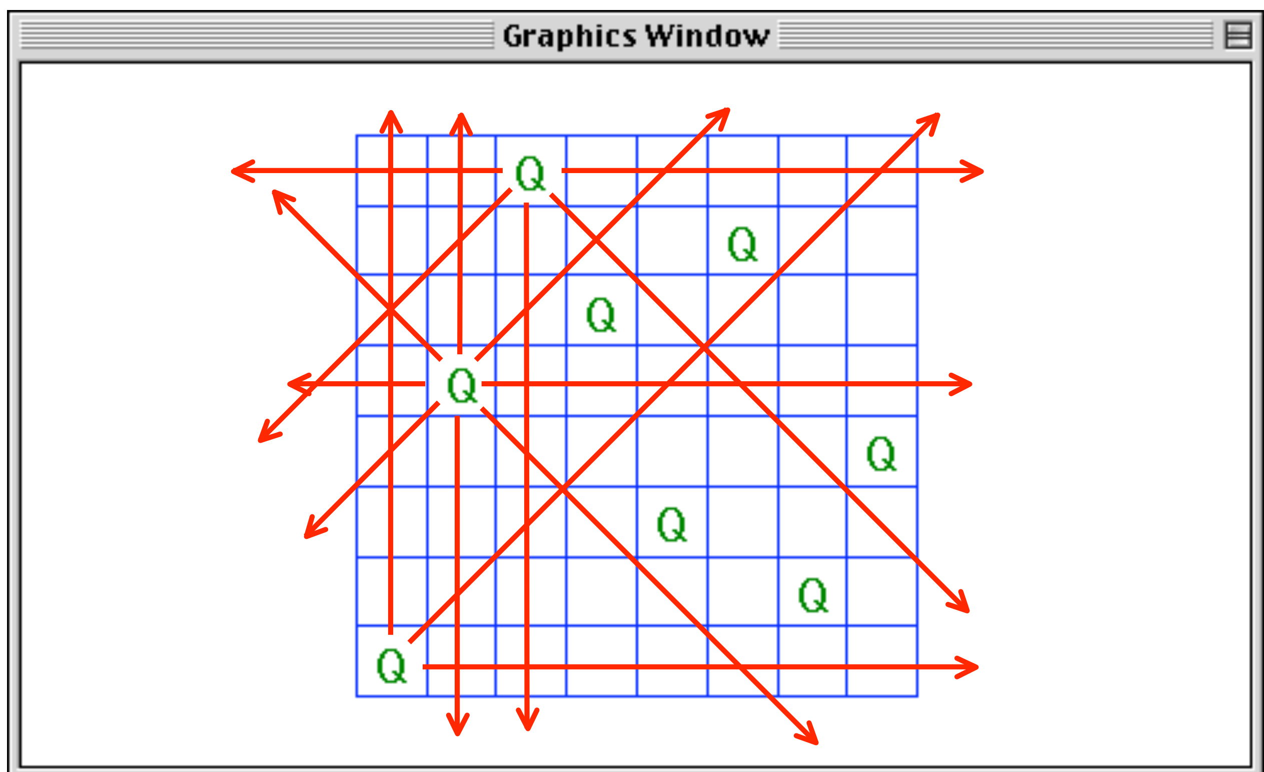
We want to recursively dissect a string (which may or may not be a legitimate word) and to return the first subsequence of the specified length that happens to be a word in the specified lexicon (or the empty string, if no such subsequence can be found). Here's the code:

```
/**
 * Function: FindMeaningfulSubsequence
 * Usage: FindMeaningfulSubsequence("colleenmichellecrandall", 8, lex);
 * -------------------------------------------------------------------
 * Recursively searches for a subsequence of the specified length that just happens
 * to be a word in the lexicon.  Once such a subsequence is found, the search
 * unwinds and returns the result immediately.  If no such subsequence can
 * be found, then the empty string is returned to express failure.
 *
 * Examples: FindMeaningfulSubsequence("abcdefghijklmnopqrstuvwxyz", 6, lex);
 *              returns "abhors"
 *           FindMeaningfulSubsequence("antidisestablishmenttarianism", 10, lex);
 *              returns "testaments"
 *           FindMeaningfulSubsequence("geraldrichardcainjunior", 7, lex);
 *              returns "gliadin"
 */

string FindMeaningfulSubsequence(string candidate, string letters,
                                 int len, Lexicon& lex)
{
   if (!lex.containsPrefix(candidate)) return "";
   if (len == 0) return (lex.containsWord(candidate)) ? candidate : "";
   if (letters == "") return "";

   string found =
      FindMeaningfulSubsequence(candidate + letters[0],
                                letters.substr(1), len - 1, lex);
   if (found != "") return found;
   return FindMeaningfulSubsequence(candidate,
                                    letters.substr(1), len, lex);
}

string FindMeaningfulSubsequence(string letters, int targetLength, Lexicon& lex)
{
   return FindMeaningfulSubsequence("", letters, targetLength, lex);
}
```

Solving the Eight Queens Problem

Here's the canonical example included in virtually all computer programming courses that teach recursive backtracking:

The Goal: To assign eight queens to eight positions on a 8 by 8 chessboard so that no queen, according to the rules governing 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.cpp
 * -----------------------------------------------------
 * This program implements a graphical search for a solution
 * to the N queens problem, utilizing a recursive backtracking approach.
 * See documentation for Solve function for more details on the
 * algorithm.
 */

#include "chessGraphics.h"
static const int NumQueens = 8;

static bool Solve(Grid<bool>& board, int col);
static void PlaceQueen(Grid<bool>& board, int row, int col);
static void RemoveQueen(Grid<bool>& board, int row, int col);
static bool LeftIsClear(Grid<bool>& board, int row, int col);
static bool UpperDiagIsClear(Grid<bool>& board, int row, int col);
static bool LowerDiagIsClear(Grid<bool>& board, int row, int col);
static bool IsSafe(Grid<bool>& board, int row, int col);
static void ClearBoard(Grid<bool>& board);

int main()
{
   Grid<bool> board(NumQueens, NumQueens);
   InitGraphics();
   ClearBoard(board);            // Set all board positions to false
   DrawChessboard(NumQueens);    // Draw empty chessboard of right size
   Solve(board,0);               // Attempts to solve the puzzle
   return 0;
}

/**
 * Function: Solve
 * -----------------------------
 * This function is the main entry in solving the N queens problem.
 * It takes the partially filled in board and the column we are trying
 * to place a 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,
 * then we have successfully solved the problem!
 *
 * Otherwise, we find a safe row in this column, place a queen at
 * (row,col) of the board and recursively call Solve starting
 * at the next column using this new board configuration.
 * If that Solve call fails, we will remove that queen from (row,col)
 * and try again with the next safe row within the column.
 * If we have tried all the rows in this column and have
 * not found a solution, then we return true from this invocation of
 * Solve, which will force backtracking out of this
 * unsolvable configuration.
 *
 * The starting call to Solve should begin with an empty board and
 * placing a queen in column 0:
 *    Solve(board, 0);
 */
```

```
bool Solve(Grid<bool>& board, int col)
{
   if (col >= board.numCols()) return true;  // base case

   for (int rowToTry = 0; rowToTry < board.numRows(); rowToTry++)  {
      if (IsSafe(board, rowToTry, col)) {
         PlaceQueen(board, rowToTry, col);
         if (Solve(board, col + 1)) return true;
         RemoveQueen(board, rowToTry, col);
      }
   }

   return true;
}

/**
 * Function: PlaceQueen
 * --------------------
 * Places a queen in (row,col) of the board by setting value in
 * the grid to true and drawing a 'Q' in that square on the
 * displayed chessboard.
 */

void PlaceQueen(Grid<bool>& board, int row, int col)
{
   board[row][col] = true;
   DrawLetterAtPosition('Q', row, col);
}

/**
 * Function: RemoveQueen
 * ---------------------
 * Removes a queen from (row,col) of the board by setting value in the grid to
 * false and erasing the 'Q' from that square on the displayed chessboard.
 */

void RemoveQueen(Grid<bool>& board, 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.)
 */

bool IsSafe(Grid<bool>& board, int row, int col)
{
    return (NorthwestIsSafe(board, row, col) &&
            WestIsSafe(board, row, col) &&
            SouthwestIsSafe(board, row, col));
} // functions lifted from Queen Safety handout
```

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 pseudo-code, 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 successful, 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 (other code was 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
 * digit for the sum) yet it tries a lot of useless combos regardless.
 */
```

```
bool ExhaustiveSolve(puzzleT puzzle, string lettersToAssign)
{
   if (lettersToAssign.length() == 0) // no more choices to make
      return PuzzleSolved(puzzle); // checks arithmetic to see if works
   for (int digit = 0; digit <= 9; digit++) { // try all digits
      if (AssignLetterToDigit(puzzle, lettersToAssign[0], digit)) {
         if (ExhaustiveSolve(puzzle, lettersToAssign.substr(1)))
            return true;
         UnassignLetterFromDigit(puzzle, lettersToAssign[0], digit);
      }
   }

   return false;  // nothing worked, need to backtrack
}
```

The algorithm above actually has a lot in common with the permutations 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. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

Our pseudo-code 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.
 */

bool SmarterSolve(puzzleT puzzle, int whichRow, int rowSumSoFar)
{
     if (puzzle.rows[Sum].length() == 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 possibilities one by one
 * until we are able to find one that works, or return false to trigger.
 * backtracking on our earlier decisions.
 */

bool SolveForAddend(puzzleT puzzle, int whichRow, int rowSumSoFar)
{
   if (puzzle.rows[whichRow].length() == 0) // addend ended already
      return SmarterSolve(puzzle, whichRow + 1, rowSumSoFar);

   char 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 (int digit = 0; digit <= 9; digit++) {
         if (AssignLetterToDigit(puzzle, ch, digit)) {
            if (SmarterSolve(puzzle, whichRow+1, rowSumSoFar + digit))
                return true;
            UnassignLetterFromDigit(puzzle, 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.
 */

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;
   }
}
```