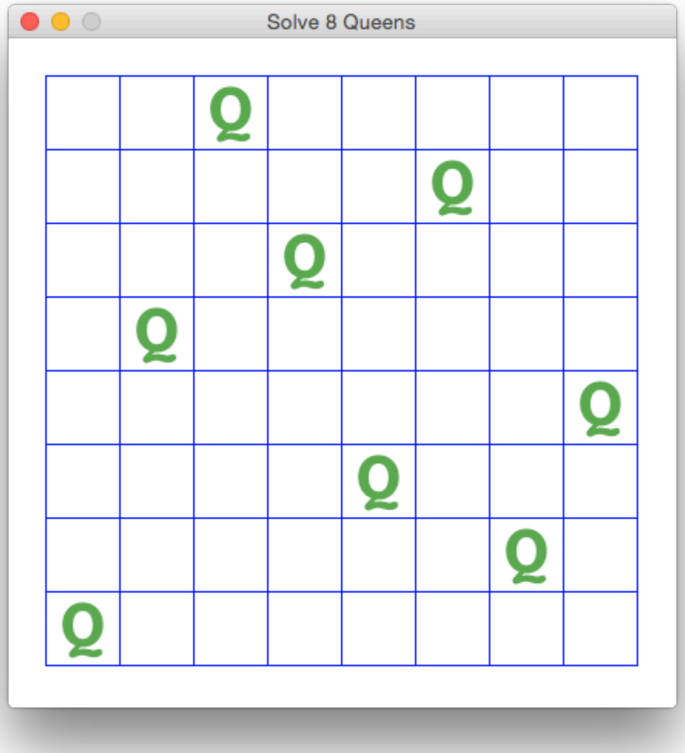
CS106X
Autumn 2019

Handout 13
October 9, 2019

Recursive Backtracking II

Solving the Eight Queens Problem

The Eight Queens Problem is a classic programming puzzle that asks whether it's possible to place eight queens on an 8 x 8 chessboard in such a way that they can all coexist without attacking each other. Placing nine queens on an 8 x 8 is impossible—there's a pigeonhole principle argument against it, for at least two queens would always need to occupy the same column. But it's not immediately obvious whether eight queens can be placed on an 8 x 8 board, nor is it obvious whether N queens can be placed on an N x N board in general.



		Q					
					Q		
			Q				
	Q						
							Q
				Q			
						Q	
Q							

One approach—by far the most common programmatic one I know of—uses recursive backtracking to discover a solution, and that approach is spelled out on the next page:

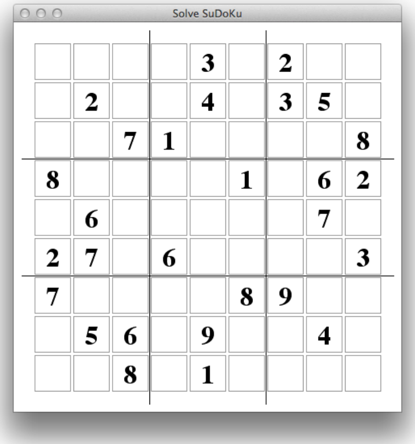
```
static bool solve(QueensDisplay& display, Grid<bool>& board, int col) {

   if (col == board.numCols())return true;

   for (int rowToTry = 0; rowToTry < board.numRows(); rowToTry++) {
      display.considerQueen(rowToTry, col);
      if (isSafe(board, rowToTry, col)) {
         board[rowToTry][col] = true;
         display.provisionallyPlaceQueen(rowToTry, col);
         if (solve(display, board, col + 1)) {
            display.permanentlyPlaceQueen(rowToTry, col);
            return true;
         }
         board[rowToTry][col] = false;
      }
      display.removeQueen(rowToTry, col);
   }

   return false;
}

static void solve(QueensDisplay& display, Grid<bool>& board) {
   solve(display, board, 0);
}
```

The second of the two versions is called on an empty board, and the first one implements the recursive backtracking. Each call to **solve** assumes that queens have been placed in columns 0 through **col – 1** in a configuration that allows them all to coexist peacefully. The **solve** call systemically searches its own column for a row where yet another queen can be placed without introducing a conflict, and then recurs on **col + 1**. If the recursive call on **col + 1** returns **true**, then that **true** is immediately propagated up to whoever called us. If it returns **false**, we backtrack by lifting the queen we placed and advancing on to higher rows. Only when **solve** has tried to extend the partial solution it inherited in every way possible—and failed every time—does it return **false**.

Solving SuDoKu Puzzles [idea by Julie Zelenski]

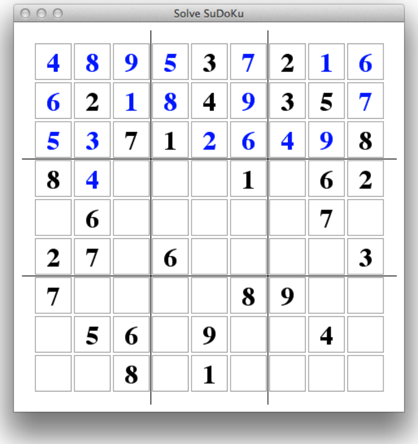
Recursive backtracking can also be used to solve SuDoKu puzzles by systematically considering every single way to legitimately place a number in some open square that, at least for the moment, works, and then recurring on the same board to see if that decision was a good one.

Original



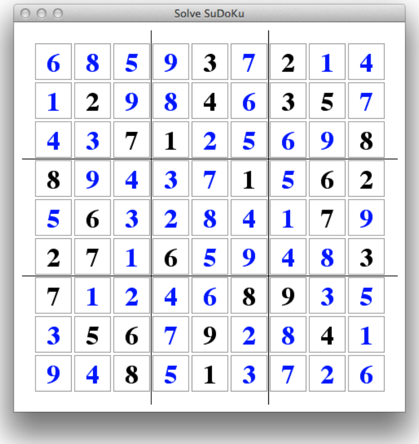
				3		2		
	2			4		3	5	
		7	1					8
8					1		6	2
	6						7	
2	7		6					3
7					8	9		
	5	6		9			4	
		8		1				

Mid Progress



4	8	9	5	3	7	2	1	6
6	2	1	8	4	9	3	5	7
5	3	7	1	2	6	4	9	8
8	4				1		6	2
	6						7	
2	7		6					3
7					8	9		
	5	6		9			4	
		8		1				

Solved



6	8	5	9	3	7	2	1	4
1	2	9	8	4	6	3	5	7
4	3	7	1	2	5	6	9	8
8	9	4	3	7	1	5	6	2
5	6	3	2	8	4	1	7	9
2	7	1	6	5	9	4	8	3
7	1	2	4	6	8	9	3	5
3	5	6	7	9	2	8	4	1
9	4	8	5	1	3	7	2	6

```
static bool solve(SuDoKuDisplay& display, Grid<int>& board) {
   int row, col;
   if (!findLocation(board, row, col)) return true;

   for (int digit = 1; digit <= 9; digit++) {
      if (isLegal(board, row, col, digit)) {
         board[row][col] = digit;
         display.provisionallyPlaceNumber(row, col, digit);
         if (solve(display, board)) {
            display.permanentlyPlaceNumber(row, col);
            return true;
         }
         board[row][col] = kEmpty;
         display.liftNumber(row, col);
      }
   }

   return false;
}
```

For those new to SuDoKu, the challenge is to fill in all empty squares with numbers 1 through 9 so that each digit appears exactly once per row, once per column, and once per 3 x 3 block. There is no denying the above is classic recursive backtracking—even if it's very brute force and not very intelligent.

`isLegal` decides, given the current state of the board, whether **`digit`** can be placed at the identified position without violating the rules. The suite of **`SuDoKuDisplay`** methods update the graphics window to convey whether we're considering, committing to, or abandoning some choice. The only function students find confusing is **`findEmptyLocation`**. From context, it appears to return a **`true`** if and only if there's some unassigned slot, but what isn't clear is that, when **`true`** is returned, **`row`** and **`col`** are updated (by reference) to some empty location's coordinates. It becomes clearer if you see the code for it, so here it is:

```
static const int kEmpty = 0;
static bool findLocation(const Grid<int>& board, int& row, int& col) {
   for (row = 0; row < board.numRows(); row++) {
      for (col = 0; col < board.numCols(); col++) {
         if (board[row][col] == kEmpty) return true;
      }
   }

   return false;
}
```

This particular implementation just searches top-to-bottom, left-to-right until it finds something empty. It's fairly naïve and results in a solution that takes its time for all but the most trivial of boards. However, it's possible to search not just for any empty square, but for the empty square that is more constrained than any other. We can use the **`isLegal`** routine to brute-force double-**`for`** loop over all locations, keeping track of the location offering the smallest number of options. There's no sense, for instance, fussing over all of the empty cells in the upper left corner of the board if there's some cell in the lower right that can only be assigned one number.

```
static const int kNumDigits = 9;
static int countNumOptions(const Grid<int>& board, int row, int col) {
   int numOptions = 0;
   for (int digit = 1; digit <= kNumDigits; digit++) {
      if (isLegal(board, row, col, digit))
         numOptions++;
   }
   return numOptions;
}

static bool findLocation(const Grid<int>& board, int& row, int& col) {
   int smallestNumOptions = kNumDigits + 1;
   for (int r = 0; r < board.numRows(); r++) {
      for (int c = 0; c < board.numCols(); c++) {
         if (board[r][c] == kEmpty) {
            int numOptions = countNumOptions(board, r, c);
            if (numOptions < smallestNumOptions) {
               row = r;
               col = c;
               smallestNumOptions = numOptions;
            }
         }
      }
   }

   return smallestNumOptions <= kNumDigits;
}
```

Regular Expressions and String Matching [prose by Jerry Cain]

A regular expression—or regex, for short—is a **`string`** used to pattern match words in the English language. The simplest regular expressions consist of just lowercase letters, but they're also allowed to contain one or more **character sets** like **`[a-z]`**, and the presence of **`[a-z]`** in a regular expression matches any lowercase letter. Here're a few examples of regular expressions and the English words that match them:

regex	*matches*
`and`	and
`[a-z]lur`	blur, slur
`wil[a-z]`	wild, wile, wili, will, wily, wily
`m[a-z][a-z]m`	maim, malm, marm, mumm
`x[a-z][a-z][a-z]x`	xerox
`[a-z]x[a-z]`	axe, exo, oxo, oxy

The notion of a character set can be generalized to specify one or more smaller ranges to represent sets of lowercase letters, as with:

character set	*possible characters*
`[a-g]`	**`abcdefg`**
`[c-gmw-z]`	**`cdefgmwxyz`**
`[aeiou]`	**`aeiou`**
`[x-za-bp]`	**`abpxyz`**

Note that isolated characters can sit among zero or more ranges to compactly express a small set of characters, as I do with the three of the four sample character sets above. This notation allows us to match a more constrained set of English words:

regex	*matches*
`m[aeiou][x-z]`	max, may, mix, miz, moy, moz, mux
`z[a-cor-z][a-gkn-p]`	zag, zap, zoa, zoo
`[a-c][d-g][h-m][n-q][r-z]`	adios, agios, aglow, below

Finally, an asterisk—i.e., one of these things: `'*'`—can follow any character or character set as an instruction that the single character or character set preceding it can be skipped and go unmatched, be matched exactly once, or be matched an arbitrarily large number of times.

What can regexes look like now, and what strings do they match? Here are some examples:

- **`aa[a-z]*`** matches all those words that begin with aa, including *aa*, *aah*, *aahed*, *aardvark*, *aardvarks*, and *aasvogel*. The **`[a-z]*`** portion of **`aa[a-z]*`** can match the empty string, a single letter, or an arbitrary string of length 2 or more.
- **`[a-z]*zz[a-z]*`** matches all of those words that contain a zz somewhere, including *buzz*, *jazziest*, *puzzle*, *sizzle*, *snazzy*, and *zyzzyvas*.
- **`[a-g]*`** matches all those words that can be formed using just the first seven letters of the alphabet, including *begged*, *cabbage*, *deface*, *defaced*, *feedbag*, and *gaffed*. Musicians love these words, because they can be formed using just the notes of a C major scale.
- **`[aeiou][aeiou][aeiou][aeiou]*`** matches all of the English words of length 3 or more that contain only the five principal vowels. Only one word matches: *eau*. Gotta love the French.
- **`[a-z]*a[a-z]*e[a-z]*i[a-z]*o[a-z]*u[a-z]*y[a-z]*`** matches the six words that contain all six vowels (this time counting y) where a, e, i, o, u, and y appear in that order. Congratulations to *abstemiously*, *adventitiously*, *facetiously*, and *sacrilegiously* for being part of this distinguished set.

For this exercise, we'll work through the decomposition of a recursive backtracking function called **`matches`** that decides whether a regex matches a string of lowercase letters (presumably a word in the English language).

Our implementation of **`matches`** will benefit from a helper function called **`expand`**, which takes a single character set and returns a sorted string of all of the lowercase letters it expands to, as with:

set	expand(set)
[a-g]	abcdefg
[x-ya-g]	abcdefgxy
[a-empw-z]	abcdempwxyz
[aeiou]	aeiou
[a-ea-ed-fa-eeeee]	abcdef

Our implementation needs to handle redundancies like those you see in the last example above, and the string of lowercase letters returned should be sorted in lexicographic order. We'll assume that the first character is always **`'['`**, the last character is always **`']'`**, there's at least one character between the **`'['`** and the **`']'`**, and that the character set identifies only lowercase letters and is otherwise well formed.

(This part doesn't involve recursion, but it's fun to implement anyway, since we can only assume it'll help out with the implementation of **`matches`**.)

```
static string expand(const string& set) {
   Set<char> charset;
   int i = 1;
   while (i < set.size() - 1) {
      char low = set[i];
      char high = set[i + 1] == '-' ? set[i + 2] : low;
      i += set[i + 1] == '-' ? 3 : 1;
      for (char ch = low; ch <= high; ch++) charset += ch;
   }

   string expanded;
   for (char ch: charset) expanded += ch;
   return expanded;
}
```

We'll also benefit from a second helper function called **split**, which takes a nonempty regular expression and pulls off the portion that might be matched by a word's first character. We'll code to this interface:

```
static void split(const string& regex, string& first,
                  bool& starred, string& rest);
```

Assuming that **first** and **rest** are **string**s and **starred** is a **bool**, the following illustrates how **first** and **rest** should be populated for the provided regexes:

regex	split(regex, first, starred, rest)
awxyz	**first** gets "**a**", **starred** gets **false**, **rest** gets "**wxyz**"
[ae]*w*	**first** gets "**[ae]**", **starred** gets **true**, **rest** gets "**w***"
z	**first** gets "**z**", **starred** gets **false**, **rest** gets ""
z*	**first** gets "**z**", **starred** gets **true**, **rest** gets ""

To be clear, **starred** is populated with **true** if any only if the leading portion placed in **first** is optional and repeatable, and **rest** is populated with everything beyond **first** and, if present, the companion *.

```
static void split(const string& regex, string& first,
                  bool& starred, string& rest) {
   int pos = !isalpha(regex[0]) ? regex.find(']') + 1 : 1;
   first = regex.substr(0, pos);
   starred = pos < regex.size() && regex[pos] == '*';
   if (starred) pos++;
   rest = regex.substr(pos);
}
```

Finally, using our **expand** and **split** functions, we can more easily implement **matches**, which uses recursive backtracking to decide whether the supplied regex matches the supplied word. Because backtracking is required, you should only make as many recursive calls as needed in order to produce a **true** or **false**. (Code is on the next page!)

```
static bool matches(const string& regex, const string& word) {
   if (regex.empty()) return word.empty();

   bool starred;
   string first, rest;
   split(regex, first, starred, rest);

   string set = first.size() == 1 ? first : expand(first);
   if (word.empty() || set.find(word[0]) == string::npos) {
      return starred && matches(rest, word);
   }

   // consider first character being consumed by first
   if (matches(rest, word.substr(1))) return true;
   if (!starred) return false;
   return matches(rest, word) || matches(regex, word.substr(1));
}
```

There are four ways we can arrive at another call to **`matches`**.

- First, if **`word`** is empty or if **`word[0]`** doesn't match the **`first`** surfaced by **`split`**, then we would only match if the pattern is **`starred`** (allowing for zero occurrences of **`first`**) and if the **`rest`** matches the full **`word`**.
- A second option should the first not work out: match **`word[0]`** to **`first`** (we know it does, else we wouldn't have gotten this far) and see if **`rest`** aligns with **`word.substr(1)`**.
- Options three and four exist, but only if **`first`** was **`starred`**: We consume the leading character of **`word`** but keep **`first`** in **`regex`** (exercising the option that `'*'` allows for two or more matches) or skip **`first`** entirely and try to match **`rest`** to the full **`word`**.