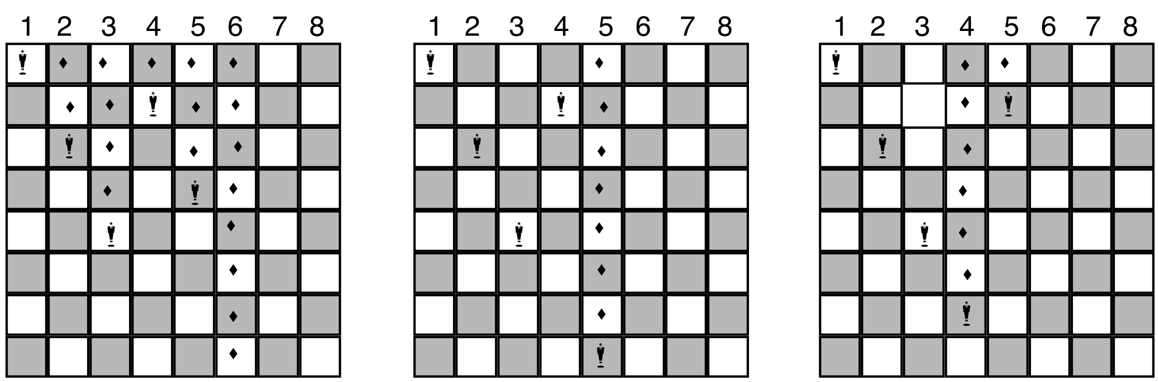
Chapter 5-1, Lecture notes

Backtracking

1. You can use recursion for problem solving.
   1.1 One method would be to take a "guess" and it is not correct then "backtrack" and try another guess.
   1.2 You can combine recursion and backtracking.

2. One classic problem is the 8 queens on one chessboard.
   2.1 The queen can attack any other piece diagonally or up and down.
   2.2 Can you put 8 queens on the board in which a queen does NOT have a clear shot at another queen?
      2.2.1 There are C(64, 8) combinations that are 4,426,165,368 ways to arrange 8 queens (or "guesses").
      2.2.2 However, we could cut down the combinations by stating you cannot place a queen on the same row or column with another queen.
      2.2.3 That cuts down of attack via row or column so the combinations is a more manageable size of 40,320 where you must check via diagonals.
      2.2.4 You can further remove some guesswork when placing the queens.
         2.2.4.1 Put the first queen in the first row and first column.
         2.2.4.2 The second queen must be placed in the second row and cannot be in the first column (column attack) nor the second column (diagonal attack).
         2.2.4.3 Figure 5-1 on page 248 shows placing 5 queens like so:



(a) (b) (c)

      2.2.5 If you cannot place a queen anywhere in column 6 then you back up and move the queen in column 5.
         2.2.5.1 Now place the queen in column 6 again.
         2.2.5.2 But if you still can't place the queen in column 6 and have gone through all the rows, you must backup all the way to column 4, move its queen, and start over with column 5 then column 6.

2.3 The pseudo code to place the queens is:

```
placeQueens (in queenPtr : Queen)
// places queens in eight columns

    if (queen's column is greater than the last column)
        problem is solved

    else
    {   while (unconsidered squares exist in queen's column and the problem
                is unsolved)
        {   Determine the next square in queen's column that is not under
                    attack by a queen in an earlier column
            if (such a square exists)
            {   Place a queen in the square
                // try next column
                placeQueens(create queen(firstRow, queen's column + 1))
                if (no queen is possible in the next column)
                {   Delete the new queen
                    Remove the last queen placed on the board and
                        Consider the next square in that column
                } // end if
            } // end if
        } // end while
    } // end if
```

2.4 To start the whole thing going, the doEightQueens routine is called like so:

```
doEightQueens()
{   placeQueens(newQueen(firstRow, firstColumn)
} // end doEightQueens
```

3 The easiest way to backtrack would be to use the STL class of vector.

3.1 The vector can use a push_back routine to place an item at the end of a list and use the pop_back to remove an item from the end.
3.2 You can also index the vector like an array only the vector will dynamically grow and shrink (and has a "size" command to tell how many items are there).
3.3 This is ideal for "backtracking" where you place queens onto the vector and remove them if they don't work out.
3.4 The vector class has a lot of operations. The ones of interest are:

```
template <typename T> class std::vector
{
public:
	/** Default constructor
	 *  @pre  None
	 *  @post  An empty vector exists */
	vector();

	/** Creates a vector with n elements
	 * @pre  None
	 * @post  A vector of n elements exists
	 * @param n   The number of elements this vector should have */
	vector(size_type n);

	/** Determine whether the vector is empty
	 * @pre  None
	 * @post  None
	 * @return   True if the vector is empty, otherwise returns false
	bool empty() const;

	/** Determines the length of the vector. The return type size_type
	 *      is an integral type
	 * @pre  None
	 * @post  None
	 * @return  The number of items that are currently in the vector */
	size_type size() const;

	/** Inserts a new element at the end of the vector
	 * @pre  None
	 * @post  The new element is the last element in the vector
	 * @param val    The item to append onto the vector */
	void push_back(const T& val);

	/** Removes the last element at the end of the vector
	 * @pre  There is at least one element in the vector
	 * @post  The last element of the vector is removed */
	void pop_back();


	/** Removes element at i
	 * @pre  The iterator must be initialized
	 * @post  The element i pointed to is no longer in the vector
	 * @return  An iterator to the item following the removed item */
	iterator erase(iterator i);
```

```
/** Erases all the elements in the vector
 * @pre    None
 * @post  The vector has no elements */
void clear();

/** Returns an iterator to the first element in the vector
 * @pre    None
 * @post  None
 * @return  If the vector is empty, the value returned by end() is returned */
iterator begin();

/** Returns an iterator to test for the end of the vector
 * @pre    None
 * @post  None
 * @return  The value for the end of the vector was returned */
iterator end();
}; // end std::vector
```

4 The header for the Queens (“queen.h”) and for the Board (“board.h”) are under the course materials in Blackboard/WebCT under queens.
   4.1 There was some problems in trying to make a static board for all the queens.
   4.2 That was abandoned and the few routines that the board will call to a queen object will pass the pointer to the board.
   4.3 However, the placeQueens routine is:

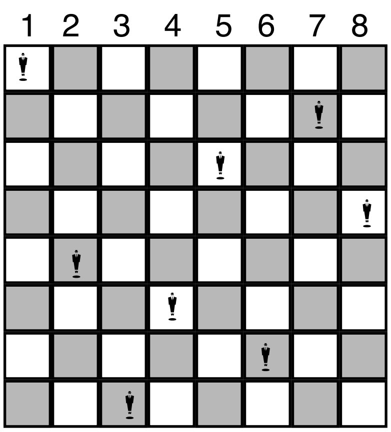
```
/** Attempts to place queens on board, starting with the designated queen */
bool Board::placeQueens(Queen *queenPtr)
{
  // Base case. Trying to place Queen in a non-existent column
  if (queenPtr->getCol() >= BOARD_SIZE)
  {  delete queenPtr;
     return true;
  } // end if

  bool isQueenPlaced = false;

  while (!isQueenPlaced && queenPtr->getRow() < BOARD_SIZE)
  { // If the queen can be attacked, ten try moving it to the next row
    // in the current column
    if (queenPtr->isUnderAttack(this))
        queenPtr->nextRow();
    // Else put this queen on the board and try putting a new queen in
    // the first row of the next column
    else
    {  setQueen(queenPtr);
       Queen *newQueenPtr = new Queen(0, queenPtr->getCol() + 1);
       isQueenPlaced = placeQueens(newQueenPtr);
       // If it wasn't possible to put the new queen in the next column,
       // backtrack by deleting the new queen and removing the last queen
       // placed and moving it down one row
```

```
        if (!isQueenPlaced)
        {  delete newQueenPtr;
           removeQueen();
           queenPtr->nextRow();
        } // end if
     } // end if
   } // end while
   return isQueenPlaced;
} // end placeQueens
```

5 The “isAttacked” routine used x and y variables to go in four diagonals and the display routine created a temporary character array to place the queens into then printed the array.
   - 5.1 The main routine is very simple: create a board object, call doEightQueens, and print the board (display).
   - 5.2 The solution it comes up with is (figure 5-2 page 255):



Chapter 5-2, Defining Languages

1 A language is nothing more than a set of symbols from a finite alphabet. You could define C++ as:

C++ Programs = {strings w : w is a syntactically correct C++ program}

1.1 So, all programs are strings but not all strings are programs.
   - 1.1.1 The C++ compiler will determine if the string belongs to the set of C++ programs.
   - 1.1.2 The term “language” does not necessarily mean a programming language or even a communication language.
   - 1.1.3 You can also have the set of algebraic expressions as a language, like so:

AlgebraicExpressions = { w: w is an algebraic expression }

2 To form a proper string for a language, you need a set of rules or a “grammar”.

2.1 The "recognition algorithm" would recognize the string as a part of the language or not depending on the grammar.
2.2 Let's try a subset of the C++ language to work with (the entire C++ grammar is a bit complex).

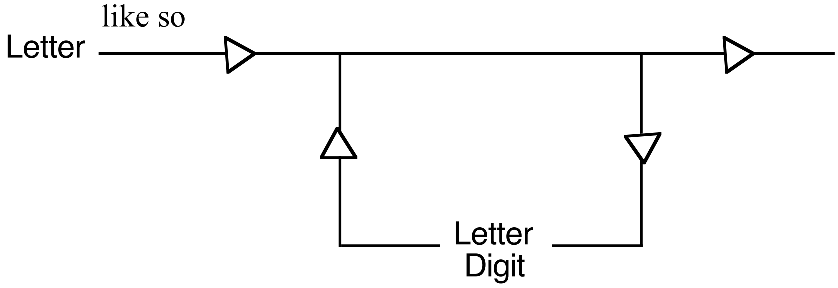
3 The Basics of Grammar are:

- x | y means x or y
- x y means x is followed by y (could also use the notation of x ● y with the "●" meaning concatenation)
- < word > means any instance of word that the definition defines

3.1 For instance, let's define C++ identifiers as a language. Grammar would be:

C++Ids = { w : w is a legal C++ identifier }

3.2 You could use a syntax diagram to represent exactly what a legal C++ identifier is like so



3.3 Or you can use something more concrete for a algorithm like so:

<identifier> = <letter> | <identifier> <letter> | <identifier> <digit>
<letter> = a | b | . . . | z | A | B | . . . | Z | _
<digit> = 0 | 1 | . . . | 9

3.4 Note that the grammar for identifier has identifier included in it – this grammar is recursive as are many grammars.
3.5 The pseudocode for whether a string id a C++ identifier would be:

```
isId(in w:string):boolean
// Returns true if w is a legal C++ identifier; otherwise returns false
if (w is of length 1) // base case
        if (w is a letter)
                return true
        else
                return false

else if (the last character of w is a letter or a digit)
        return isId(w minus its last character) // point X
else
        return false
```

4 A classic problem that would fit into a grammar is the palindrome or defined like:

Palindrome = { w : w reads the same left to right as right to left }

4.1 If w is a palindrome then:

w minus its first and last characters is a palindrome

4.2 Or the first and last characters are the same. You can define it as:

<pal> = empty string | <ch> | a <pal> a | b <pal> b | . . . | Z <pal> Z
<ch> = a | b | . . . | z | A | B | . . . | Z

4.3 Note that the grammar is definitely recursive. You can write a recursive routine to do the above. Here is the pseudocode:

```
isPal(in w : string) : boolean
// Returns true if the string w of letters is a palindrome; otherwise returns false
if (w is the empty string or w is of length 1)
        return true

else if (w's first and last characters are the same letter)
        return isPal(w minus its first and last characters)

else
        return false
```

5 Another application for recursion would be equations.

5.1 For instance, a compiler would have to take the following and parse it (break it apart so that the computer can do the operations):

y = x + z * (w/k + z * (7 * 6));

5.2 The would have to deal with parenthesis and operator precedence (do the * and / operations before the + and -).

5.3 The above is also called “infix” notation in that the operations are “in line” with the variables and values. Like:

a + b

5.4 The prefix operation would be:

+ a b

5.5 In which you would have the operation first then get two variables and/or two values (the computer works well with that notation but the programmers would not like it).
5.6 There is also the "postfix" expressions like so:

a b +

5.7 Where the variables/values come first and are followed by the operations. Prefix and postfix notations are simple to describe and implement but not popular with humans.

5.8 Here is an example of prefix grammar:

<prefix> = <identifier> | <operator> <prefix> <prefix>
<operator> = + | - | * | /
<identifier> = a | b | . . . | z

6 Trying to figure out infix operations with precedence rules are complex and will not be covered in this chapter.

Chapter 5-3, The Relationship Between Recursion and Mathematical Induction

1 There is a very strong relationship between recursion and mathematical induction.
1.1 Recursion solves a problem by specifying a solution to one or more base cases and then demonstrating how to derive the solution to a problem of an arbitrary size from the solutions to smaller problems of the same type.
1.2 Mathematical induction proves a property about the natural numbers by proving the property about a base case – usually 0 or 1 – and then proving that the property must be true for an arbitrary natural number n if it is true for the natural numbers smaller than n.

2 Mathematical induction is used to prove the "correctness" of recursive algorithms.
2.1 We'll just see how this is done with the Cost of Towers of Hanoi. Given:

```
solveTowers(count, source, destination, spare)

    if (count is 1)
        Move a disk directly from source to destination

    else
    {   solveTowers(count – 1, source, spare, destination)
        solveTowers(1, source, destination, spare)
        solveTowers(count – 1, spare, destination, source)
```

} // end if

2.2 If we start with N disks, how many moves does solveTowers make to solve the problem? Well with the base case of 1, it is easy:
moves(1) = 1

2.3 But with N > 1, it is not so easy.
2.4 So, looking at the algorithm, we can say that moving N disks (with N > 1) is:

moves(N) = moves(N – 1) + moves(1) + moves(N – 1)

2.5 So you have a recurrence relation for the number of moves for N disks:

moves(1) = 1
moves(N) = 2 * moves(N – 1) + 1 if N > 1

2.6 For example, you can determine moves (3) by:

moves(3) = 2 * moves(2) + 1
= 2 * (2 * moves(1) + 1) + 1
= 2 * (2 * 1 + 1) + 1
= 7

3 The above is an example of calculation of moves(N).
3.1 However, we would prefer "closed-form formula" (an algebraic expression) whereas you can plug in any value of N and get an answer.
3.2 You can use certain techniques to get that closed form formula but it is out of the scope of this text.
3.3 Instead, we'll pluck a formula from "thin air" and prove that it works for the Towers of Hanoi. That formula is:

moves(N) = $2^N - 1$, for all $N \geq 1$

3.4 If we plug in the value of 3 for N we will get a 7 as an answer.
3.5 But we can "prove it" by mathematical induction or establish that:

property is true for an arbitrary k ⇒ property is true for k + 1

3.6 So the "inductive hypothesis" would be that: Assume that the property is true for N = k. That is assume:

moves(k) = $2^k - 1$

3.7 The "inductive conclusion" is: Show that the property is true for N = k + 1 (i.e. show that moves(k + 1) = $2^{k+1} - 1$).
3.8 Like so:

$$
\begin{aligned}
\text{moves}(k + 1) &= 2 * \text{moves}(k) + 1 && \text{from the recurrence relation} \\
&= 2 * (2^k - 1) + 1 && \text{by the inductive hypothesis} \\
&= 2^{k+1} - 1
\end{aligned}
$$

3.9 And you have established:

property is true for an arbitrary k $\Rightarrow$ property is true for k + 1

4 That was a fairly easy proof. Not all will be that easy. However, well-structured programs are better suited to these techniques than poorly designed ones