

Michael Aiello
0266204

CS 637 Project 1
(Backtracking Programming Project)
Parts I and II

Part I

Explanation of 8 queens problems in three different programming languages.

1. Prolog (<http://www.cs.rit.edu/~ats/plcr-2003-1/html/skript-52.html>)

```
col(1). col(2). col(3). col(4). col(5). col(6). col(7). col(8).
dif(A, B, C, D, E, F, G, H) :-
    A =\= B, A =\= C, A =\= D, A =\= E, A =\= F, A =\= G, A =\= H,
    B =\= C, B =\= D, B =\= E, B =\= F, B =\= G, B =\= H,
    C =\= D, C =\= E, C =\= F, C =\= G, C =\= H,
    D =\= E, D =\= F, D =\= G, D =\= H,
    E =\= F, E =\= G, E =\= H,
    F =\= G, F =\= H,
    G =\= H.

queens(C1, C2, C3, C4, C5, C6, C7, C8) :-
    col(C1), col(C2), col(C3), col(C4), col(C5), col(C6), col(C7), col(C8),
    dif(C1, C2, C3, C4, C5, C6, C7, C8),
    dif(1+C1, 2+C2, 3+C3, 4+C4, 5+C5, 6+C6, 7+C7, 8+C8),
    dif(1-C1, 2-C2, 3-C3, 4-C4, 5-C5, 6-C6, 7-C7, 8-C8).
```

The above example exploits Prolog's built in backtracking feature. The Col clauses define the columns as small numbers from 1 to 8. Diff succeeds only if no two columns passed to it are the same. Queens succeeds only for solutions to the 8 queens problem by checking that no columns are the same in the current row, or in the row above or below it. Solutions are built upon this rule by adding one row at a time until 8 rows have been added. When failure occurs, Prolog backtracks and unifying the column to a different col and continues checking. (Note: this framework was used to solve the programming part of this project).

2. Haskell (<http://c2.com/cgi/wiki?EightQueensInManyProgrammingLanguages>)

```
boardSize = 8
queens 0 = [[]]
queens n = [ x : y | y <- queens (n-1), x <- [1..boardSize], safe x y 1]

safe x [] n = True
safe x (c:y) n = and [ x /= c , x /= c + n , x /= c - n , safe x y (n+1)]
```

Below is the same program broken down with comments by Michael Aiello

```
boardSize = 8           --create an 8 by 8 board
queens 0 = [[]]         --queens of 0 has no answers
queens n =
[
x : y |                 --split into head(x) and tail(y)
y <- queens (n-1),     --recursively find queens(n-1) and put it in tail
x <- [1..boardSize],   --create a list of [1,2,3,4,...boardsize] and unify to x
safe x y 1             --check to see if it is safe to place a queen at x in solution
]                       --y
                        --the 1 is for shifting left and right one (diagonal
                        --checking)
                        --if this fails we backtrack and try another value of x

safe x [] n = True     --placing in an empty list is safe

safe x (c:y) n =       --match x(the location for the queen)
                        --      y broken into c(head) and y(tail)
                        --      n the shift amount when checking rows above and below

and                    --make sure all of the following predicates are true
[
x /= c ,               --placement is not in the same spot as another
x /= c + n ,          --placement is not next to the head to the right
x /= c - n ,          --placement is not next to the head on the left
safe x y (n+1)        --recursively check to see if this placement is safe for
]                     --the remainder of the list (tail) n+1 (along the diagonal)

]
```

This solution is similar to the one discussed the final day of class. It takes advantage of the fact that 2 queens can not be in the same row. To start, it adds a queen to the first row and then attempts to add a queen to the second row. If this is successful if it is safe to add the queen to the second row it continues to the third and so forth. If it realizes that it is an unsafe placement, it backtracks one row and tries a different placement for the previous row and continues to check if a placement is safe. To determine if a placement is safe, the program recursively checks "up" the board moving to the right or left by one each row up (checking to make sure the newly placed queen is not in check on the diagonal on the way up).

3. C language (<http://c2.com/cgi/wiki?EightQueensInManyProgrammingLanguages> (I modified it a bit))

```
inline int calcDiag1Bit(int max, int row, int col) { return 1 << (max-col-row); }
inline int calcDiag2Bit(int max, int row, int col) { return calcDiag1Bit(max, row, max-
col); }
int solve(int max, int solutions, int row, int col, int allCols, int diag1, int diag2) {
    int rowBit, colBit, diag1bit, diag2bit;
    for (col=0; col<max; col++) {
        if (allCols & (colBit=(1 << col))) continue;
        if (diag1 & (diag1bit=calcDiag1Bit(max, row, col))) continue;
        if (diag2 & (diag2bit=calcDiag2Bit(max, row, col))) continue;
        if (row >= (max-1))
            ++solutions;
        else
            solutions = solve(max, solutions, row+1, col, allCols|colBit,
                diag1|diag1bit, diag2|diag2bit);
    }
    return solutions;
}
int main() {
    int solutions;
    solutions = solve(8, 0, 0, 0, 0, 0, 0);
    printf("%d: %d total solutions\n", 8, solutions);
}
```

Below is the same program broken down with comments by Michael Aiello

```
inline int calcDiag1Bit(int max, int row, int col) { return 1 << (max-col-row); }
//the above line declares an inline function that shifts max-col-row to the left one
//used for checking the left diagonal
inline int calcDiag2Bit(int max, int row, int col) { return calcDiag1Bit(max, row, max-
col); }
//the above line declares an inline function that shifts max-col-(max-col) to the left
//one, used for checking the right diagonal

int solve(
    int max,          //size of board
    int solutions,   //solutions counter
    int row,         //current row to check
    int col,         //current col to check
    int allCols,     //cols integer which holds (represented as binary) which col is used
    int diag1,       //diagonal left used for recursive diagonal checking
    int diag2        //diagonal right used for recursive diagonal checking
){
    int rowBit, colBit, diag1bit, diag2bit; //make local versions
    for (col=0; col<max; col++)           // for each column on the board
        {
            if (allCols & (colBit=(1 << col))) continue;
            //check for same column as previous row if failure, break out, store the
            //colBit
            if (diag1 & (diag1bit=calcDiag1Bit(max, row, col))) continue;
            //check for diagonal in previous row, if failure, break out
            if (diag2 & (diag2bit=calcDiag2Bit(max, row, col))) continue;
            //check for other diagonal in previous row, if failure, break out
            if (row >= (max-1))
                //if we have reached the number of queens to place, call it a solution
                ++solutions;
            else
                solutions =
                    solve(
                        //recursively solve
                        max,          //pass board size
                        solutions,    //solutions counter
                        row+1,        //try the next row
                        col,          //pass same column
                        allCols|colBit, //logical or all the cols with colBit
                                    //((add colBit into allCols)
                        diag1|diag1bit, //logical or the diag1 check with the
                                    //diag1bit (add diag1 into diag1bit)
                        diag2|diag2bit //logical or the diag2 check with the
                                    //diag2bit (add diag2 into diag2bit)
                    );
        }

    return solutions; //return the solution count
}

int main() {
    int solutions; //declare solutions counter
    solutions = solve(8, 0, 0, 0, 0, 0, 0); //solve for 8 queens
    printf("%d: %d total solutions\n", 8, solutions); //print solution count
}
```

This solution uses the binary representation of integers and the stack to solve the problem.

I think it can be explained best by taking a snapshot of the call stack right before identifying a solution taken on a smaller board of size 4x4 and showing the converted parameters.

```
1. solve(int max=4, int solutions=0, int row=3, int col=2, int allCols=11, int diag1=13,
int diag2=1073741830)
2. solve(int max=4, int solutions=0, int row=2, int col=0, int allCols=10, int diag1=9,
int diag2=6)
3. solve(int max=4, int solutions=0, int row=1, int col=3, int allCols=2, int diag1=8,
int diag2=2)
4. solve(int max=4, int solutions=0, int row=0, int col=1, int allCols=0, int diag1=0,
int diag2=0)
5. solve(int max=4, int solutions=0, int row=3, int col=2, int allCols=11, int diag1=13,
int diag2=1073741830)
conversion of representative parameters to binary for 5
allcols = 1011
diag1   = 1101
diag2   = 0110 ** only last 4 bits are used of the large value

6. solve(int max=4, int solutions=0, int row=2, int col=0, int allCols=10, int diag1=9,
int diag2=6)
conversion of representative parameters to binary for 6
allcols = 1010
diag1   = 1001
diag2   = 0110

7. solve(int max=4, int solutions=0, int row=1, int col=3, int allCols=2, int diag1=8,
int diag2=2)
conversion of representative parameters to binary for 7
allcols = 0010
diag1   = 1000
diag2   = 0010
```

The allCols integer has a 1 in the column if there has been a queen placed in that column the two diag integers hold a representation of the current, illegal placements of the queen, in regards to diagonal attacks. All of these parameters are passed up through the stack calls and held until a solution has been reached using them (like in this example), or an illegal placement occurs and the stack is destroyed. This continues recursively until all solutions to the 8 queens problem are reached.

4. Functional yet incomprehensible Perl in 2 lines (for fun)

```
#!/usr/bin/perl -l
sub placequeen {$_[0]=~/^(.) (.*) (??{abs$1-$3!=length$2 && 'x'})/ ? () : length $_[0]
== 8 ? @_ : map $_[0]=~$_?():placequeen("$_@"), 0..7}
print map 'x$_.'Q'.'x(7-$_)'\n', /./g for placequeen;
```

Part II (Problem 1 chosen)

A program to enumerate all integer 4×4 matrices filled with numbers so that all the sums of all possible 2 × 2 submatrices are equal to each other.

Implemented in visual prolog

Functional aspects of code with added comments (complete code attached)

```
class facts - familyDB
    trynum : (integer Number).

class predicates
    isProbl : (integer NUMBER, integer NUMBER, integer NUMBER, integer NUMBER,
              integer NUMBER, integer NUMBER, integer NUMBER, integer NUMBER,
              integer NUMBER, integer NUMBER, integer NUMBER, integer NUMBER,
              integer NUMBER, integer NUMBER, integer NUMBER, integer NUMBER) nondeterm
              anyflow.

%The letters used represent a matrix arranged in this way

%ABCD
%EFGH
%IJKL
%MNOP

clauses
    isProbl(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P):-
        trynum(A),
        A = 0,
        trynum(B), trynum(E), trynum(F),
        A <> B, A <> E, A <> F,
        B < E,
        B <> F,
        E <> F,
        A + B + E + F = 30,
        trynum(C), trynum(G),
        A <> C, A <> G,
        B <> C, B <> G,
        E <> C, E <> G,
        F <> C, F <> G,
        C <> G,
        B + C + F + G = 30,
        trynum(I), trynum(J),
        A <> I, A <> J,
        B <> I, B <> J,
        E <> I, E <> J,
        F <> I, F <> J,
        C <> I, C <> J,
        G <> I, G <> J,
        I <> J,
        E + F + I + J = 30,
        trynum(K),
        A <> K,
        B <> K,
        E <> K,
        F <> K,

        %clause takes 16 integers to match
        %Match A to a trynum() entry
        %a[0,0] = 0 *
        %Match B, E and F
        %ensure there are no duplicates
        %a[0,1]<a[1,0] *
        %check for sub matrix condition
        %Match C, G
        %ensure there are no duplicates
        %check for sub matrix condition
        %this continues for all rules
```

```

C <> K,
G <> K,
I <> K,
J <> K,
F + G + J + K = 30,
trynum(D), trynum(H),
A <> D, A <> H,
B < D, % a[0,1]<a[0,n-1] *
B <> H,
E <> D, E <> H,
F <> D, F <> H,
C <> D, C <> H,
G <> D, G <> H,
I <> D, I <> H,
J <> D, J <> H,
K <> D, K <> H,
D <> H,
C + D + G + H = 30,
trynum(M), trynum(N),
A <> M, A <> N,
B <> M, B <> N,
E < M, % a[1,0]<a[n-1,0] *
E <> N,
F <> M, F <> N,
C <> M, C <> N,
G <> M, G <> N,
I <> M, I <> N,
J <> M, J <> N,
K <> M, K <> N,
D <> M, D <> N,
M <> N,
I + J + M + N = 30,
trynum(L),
A <> L,
B <> L,
E <> L,
F <> L,
C <> L,
G <> L,
I <> L,
J <> L,
K <> L,
D <> L,
M <> L,
N <> L,
G + H + K + L = 30,
trynum(O),
A <> O,
B <> O,
E <> O,
F <> O,
C <> O,
G <> O,
I <> O,
J <> O,
K <> O,
D <> O,

```


1.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

2.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

3.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

4.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

5.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

6.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

7.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

8.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

9.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

The solution is found by matching only the newly bolded elements and ensuring that they do not repeat any previous elements before attempting to fill more of the matrix. It is a very straightforward solution for a language with built in backtracking. The order in which the rules are specified has great significance in this solution. If the checks for uniqueness are done first rather than as new elements are added, the program does not finish in a reasonable amount of time. Also adding the smallest set of connected elements that create a new, complete 2x2 submatrix eliminates some needless backtracking where checks fail because of invalid geometry. Also checks at the end of the rule set for “wraparound” conditions are not needed. This is proven below.

Assume we are given a 4x4 matrix where the sums of the elements in all possible 2x2 “non wrapping” submatrices are equal to 30(Assumption 1). We define 4 2x2 submatrices as follows

A = EFIJ

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

B=FGJK

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

C=GHKL

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

D=EIHL

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Because of the Assumption 1), we know that $A = B = C = 30$. From this, we can conclude that $A + C = 60$. The intersection of A and C is matrix B. We know that $B = 30$, if we subtract the entries in matrix B from the union of A and C, we are left with matrix D (the wraparound). We can see that $A + C - B = 30 = D$. Thus, the remaining entries in matrix d must equal 30. This same approach can be used to show that all possible wraparounds = 30. (The corners work as well once we know the initial wraparounds. $A=ABMN$, $B=BCNO$, $C=CDOP$, $D=ADMP$).

Finally, we realize that this solution it is not generalized for a matrix of size n (Not required for project). However, some thought was put into this.

In an arbitrary size matrix n by n (given n is even), it is necessary to determine the “magic subsum” for the 2x2 submatrices before beginning enumeration in order to prevent several checks to determine that 2x2 submatrices have summed to an unknown but equivalent value. The “magic subsum” using 2x2 submatrices can be found using the following equation.

$$\frac{\sum_0^{n^2-1} n}{n^2/4} = \frac{n^2 * (n^2 - 1)}{2} \bigg/ \frac{n^2}{4} = 2*(n^2-1) = \mathbf{2n^2 - 2} \text{ (Given } n = 2k \text{ where } k \text{ is a natural number)}$$

The solution would then use the result of this equation as the “magic subsum” and continue with a generalized method of adding elements to the matrix such that the sum of the values in the submatrices equal this value and there are no repeated elements.

*Note: I completed the coding of project before the “non redundant” removal check was introduced. To add these checks, I simply added the new rules when the variable to be checked against was introduced. This brought the number of solutions down from 1272 to $1272/8 = 159$.

Appendix

Example “pretty” Output followed by the compressed output for first 2 and last 2 of the 159 matrices.

Pretty

```
0    3    4    7
12   15   8   11
1    2    5    6
13   14   9   10
```

```
0    3    4    7
12   15   8   11
1    2    5    6
14   13   10   9
```

```
0    11   6    13
14   5    8    3
1    10   7    12
15   4    9    2
```

```
0    11   8    13
14   5    6    3
1    10   9    12
15   4    7    2
```

Compressed

```
0 3 4 7 12 15 8 11 1 2 5 6 13 14 9 10
0 3 4 7 12 15 8 11 1 2 5 6 14 13 10 9
0 11 6 13 14 5 8 3 1 10 7 12 15 4 9 2
0 11 8 13 14 5 6 3 1 10 9 12 15 4 7 2
```

Download Visual-Prolog here (<http://www.visual-prolog.com/vip6/download/>)