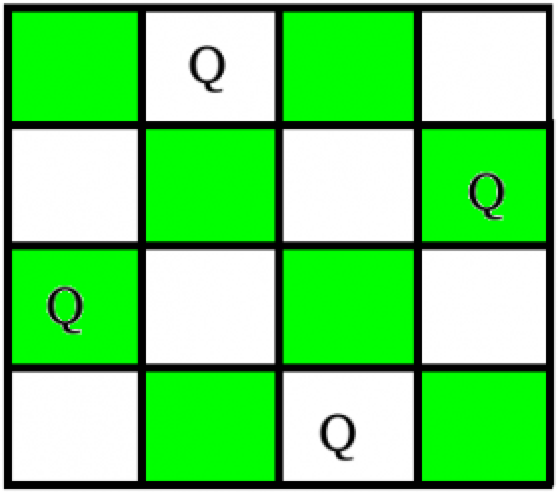
CS204

Backtracking

N-Queens Problem

- Problem: How to place N queens on an NxN chess board such that no queens can attack each other
- Rules: Queens can attack at any distance vertically, horizontally, or diagonally



Naïve Recursive Solution

- Generate all Possible placement of n queens
- Check each placement if it is following proper queen placement rule
- If a placement is proper then print that particular placement

Generate All Possible Placement

- Here all possible placement refers to all possible combination of queens position (Not permutation)
- A position of queen at ith row and jth column can be computed as $n*(i-1)+j$
- So if one possible combination in a 4 queens problem is (p, q, r, s) then (q, p, r, s) is not a separate combination.
- To achieve this, we can simply follow a strategy, 1st queen will always be positioned before 2nd queen and so on.
- So if (p,q,r,s) is a placement then $p<q<r<s$.

Generate All Possible Placement

- Place the first queen in position (1..n*n-(n-**1**)) so that you have at least n-1 positions left to place rest n-1 queens.

- Call this function recursively to place next queen which can be placed (x+1..n*n-(n-**2**)) assuming 1st (last) queen is placed at xth position.

- Continue this until all queens are placed.

Generate All Possible Placement

```
Generate (already_paced_queens, count){
Base Condition:
     if count == N // all queens are placed
     check_and_print(already_placed_queens)

for all possible placement of next queen
  Generate (already_paced_queens, count +1)
}
```

already_placed_queens: An array of positions for already placed queens

count: Number of queens already placed

Generate All Possible Placement

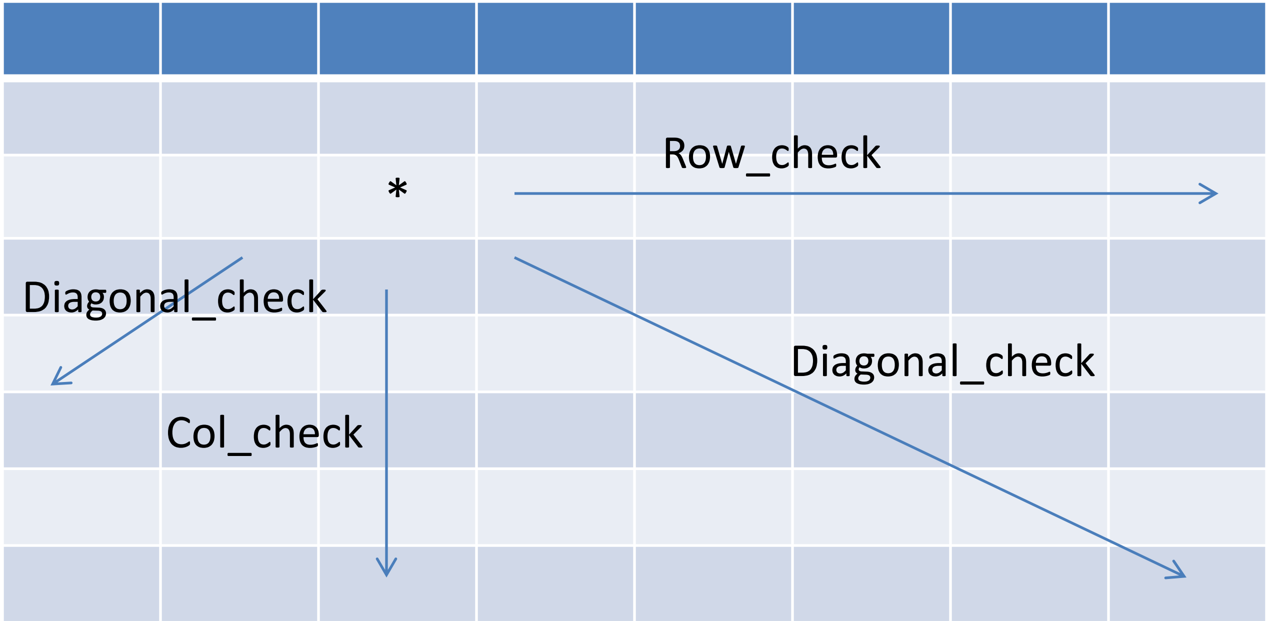
```
void generate(int x[N], int count){
int i,start;
if (count == N)
    check_and_print(x);
else {
    if(count ==0)
        start =1;
    else
        start = x[count-1] +1;
    for(i=start;i<=N*N-(N-count-1);i++){
        x[count]=i;
        generate(x,count+1);
    }

    }
}
}
```

Check And Print

- It takes an array of position of all placed queens and check if all the constraints are satisfied.
- If all constraints are satisfied then print the position.

```
void check_and_print(int x[N]){
int placed[N][N]={0},i,m,row,col;
for(i=0;i<N;i++){
        m=x[i];
        row=(m-1)/N;
        col=(m-1)%N;
        placed[row][col]=1;
        }
if (check(placed))
        print(placed);
}
```



*

Row_check

Diagonal_check

Col_check

Diagonal_check

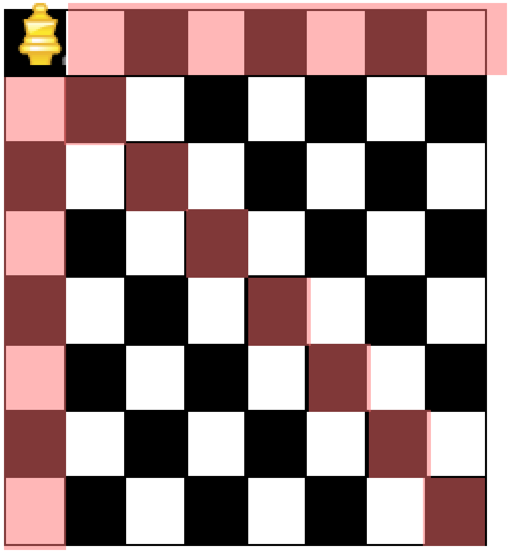
Check

```
int check(int placed[N][N]){
int i,j,k;
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        if (placed[i][j]){
            if (check_row(i,j,placed) && check_col(i,j,placed) &&
check_diagonal(i,j,placed))
                continue;
            else
                return 0;
        }
    }
}
return 1;
}
```

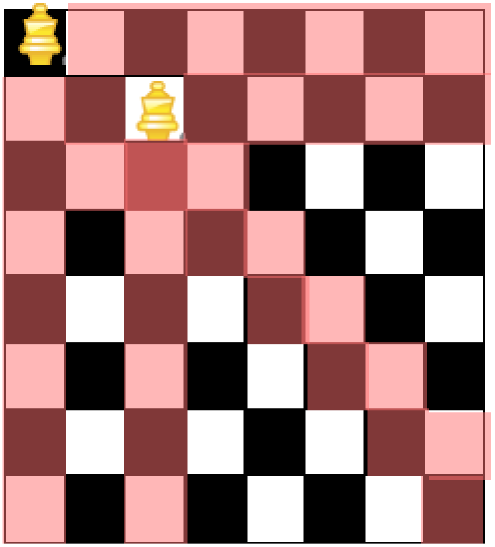
Efficiency??

Backtrack search approach

- Place 1st queen in a viable option.



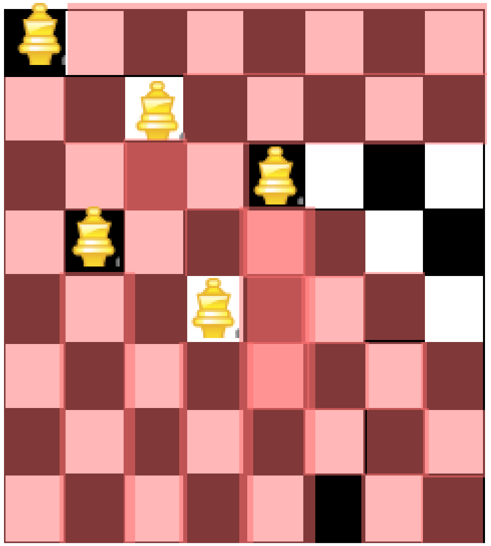
- Now place 2nd queen



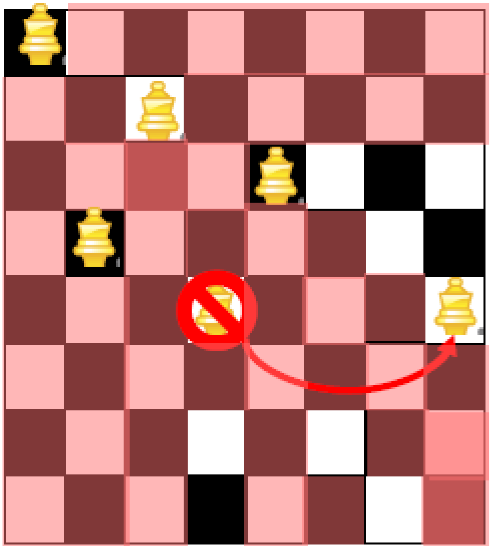
- Now place others as viable

- After this configuration here, there are no locations in row 6 that are not under attack from the previous 5

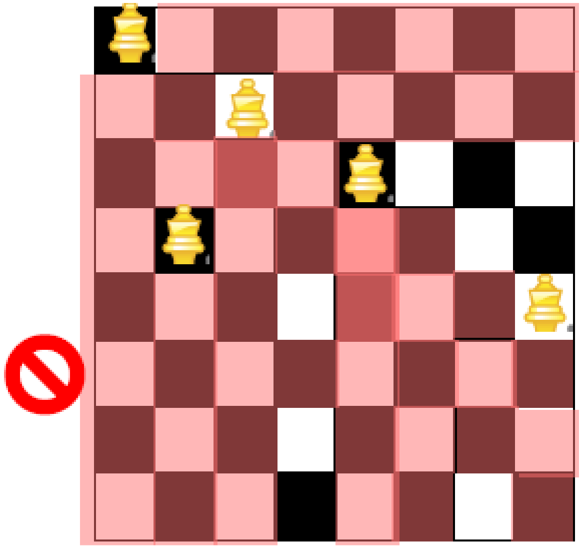
- BACKTRACK!!!



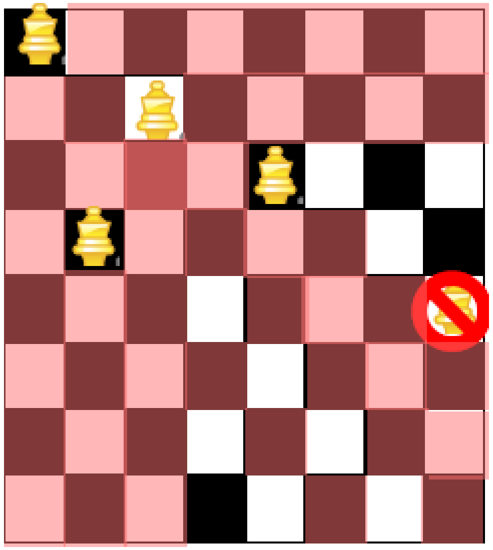
- So go back to row 5 and switch assignment to next viable option and progress back to row 6



- But still no location available so return back to row 5

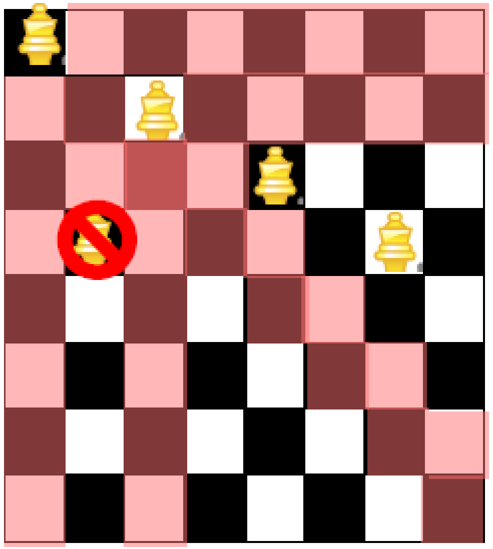


- But now no more options for row 5 so return back to row 4
- BACKTRACK!!!!



- Move to another viable place in row 4 and restart row 5 exploration

- Keep going until you successfully place row 8 in which case you can return



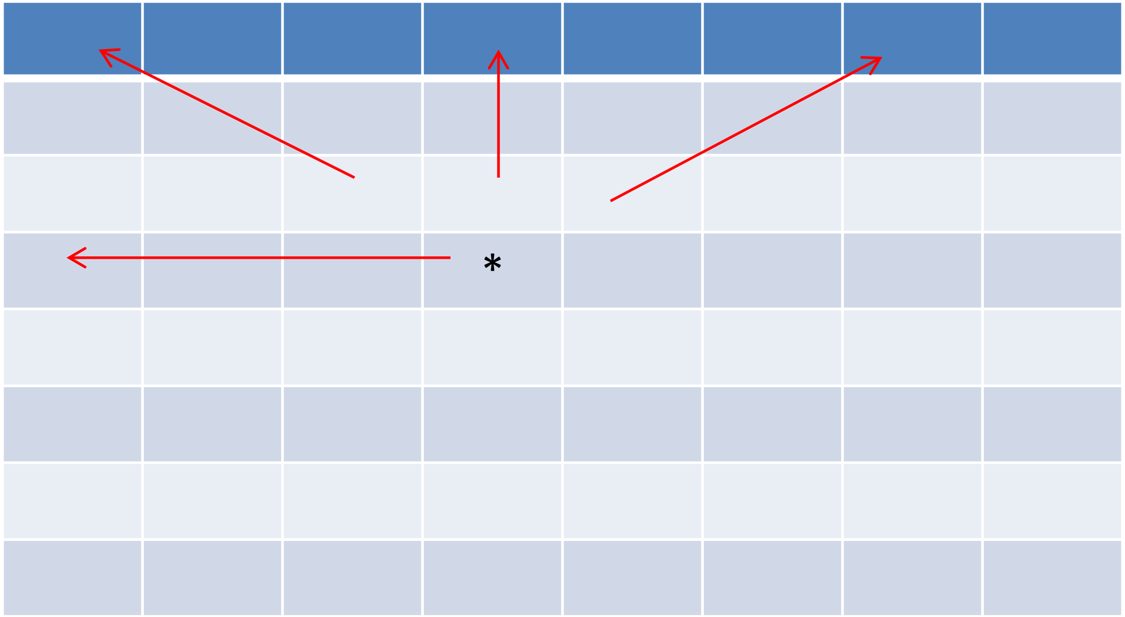
Comparison of simple recursion and backtracking

- Recursion can be used to generate all option
  - 'brute force' / test all options approach
  - Test for constraint satisfaction only at the bottom of the 'tree'
- But backtrack search attempts to 'prune' the search space
  - Rule out options at the partial assignment level

1. Start in the topmost row
2. If all queens are placed return true
3. Try all columns in the current row. Do following for every tried row.
    a) If the queen can be placed safely in this column then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
    b) If placing the queen in [row, column] leads to a solution then return true.
    c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other column.
4. If all columns have been tried and nothing worked, return false to trigger backtracking

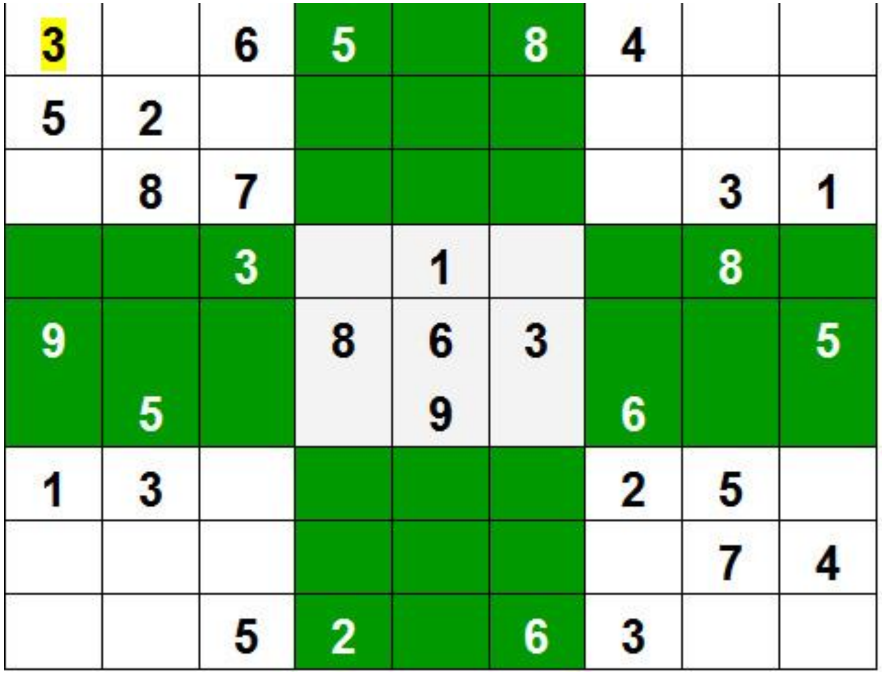
```
int solveNQ(int placed[N][N], int row){
    int col;
    if (row >= N)
        return 1;
    for (col = 0; col < N; col++) {
        if (isSafe(row, col,placed)) {
            placed[row][col]=1;
            if (solveNQ(placed,row+1))
                return 1;
            placed[row][col]=0; //backtrack
        }
    }
    return 0;
}
```

```
int isSafe(int i, int j, int placed[N][N]){
if (check_row(i,j,placed) && check_col(i,j,placed)
&& check_diagonal(i,j,placed))
return 1;
}
```



Sudoku

- Given a partially filled 9×9 2D array grid[9][9], the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.



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

Input and Output

- **Input:** grid =

```
{ {3, 0, 6, 5, 0, 8, 4, 0, 0},
  {5, 2, 0, 0, 0, 0, 0, 0, 0},
  {0, 8, 7, 0, 0, 0, 0, 3, 1},
  {0, 0, 3, 0, 1, 0, 0, 8, 0},
  {9, 0, 0, 8, 6, 3, 0, 0, 5},
  {0, 5, 0, 0, 9, 0, 6, 0, 0},
  {1, 3, 0, 0, 0, 0, 2, 5, 0},
  {0, 0, 0, 0, 0, 0, 0, 7, 4},
  {0, 0, 5, 2, 0, 6, 3, 0, 0} }
```

Output:

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

- Sudoku can be solved by one by assigning numbers to empty cells.
- Before assigning a number, check whether it is safe to assign. Check that the same number is not present in the current row, current column and current 3X3 subgrid.
- After checking for safety, assign the number, and recursively check whether this assignment leads to a solution or not.
- If the assignment doesn’t lead to a solution, then try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, return false and print no solution exists.

```
int SolveSudoku(int grid[N][N])
{
  int row, col;
   if (there is no unassigned location)   //base case
            return 1;
    else
            get row and column of any unassigned location
            for (int num = 1; num <= 9; num++) {
                    if (isSafe(grid, row, col, num)) {
                            grid[row][col] = num;
                            if (SolveSudoku(grid))
                                    return 1;
                    grid[row][col] = UNASSIGNED;
            }
  }
  return false;
}
```

Homework

- Print all possible permutation of 1,2,3,4
- Print all possible combination of 1,2,3,4