Solving 8-Queens Problem by Generating Permutations

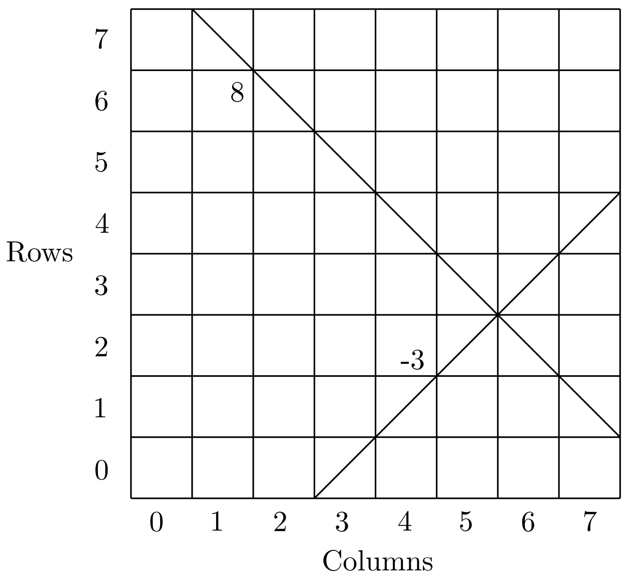
Nitin Verma
mathsanew.com

December 11, 2021

The well-known *8-Queens Problem* is to place 8 queens on an 8×8 chessboard such that no queen attacks any other queen. It is equivalent to the condition that on the row, column and the two diagonals of a queen's square, there can be no other queen.

We can identify the 8 rows and 8 columns using integers $\{0, 1, 2, \ldots, 7\}$ as shown in the figure below. Now onward we will use $\mathbb{Z}_8$ to denote the set of 8 integers: $\{0, 1, 2, \ldots, 7\}$. There are two types of diagonals: *ascending* and *descending*; the figure shows one ascending and one descending diagonal marked with -3 and 8 respectively (these identifiers are described below).

Figure 1: Numbering Rows, Columns and Diagonals of a Chessboard



Let us denote the square on row r and column c as square (r, c). Note that on any given ascending diagonal, any square (r, c) has a fixed value of $(r - c)$. For the 15 such diagonals, these values are distinct and range from -7 to 7. Similarly, on any given descending diagonal, any square (r, c) has a fixed value of $(r + c)$. For the 15 such diagonals, these values are distinct and range from 0 to 14.

Thus we can uniquely identify the 15 ascending and 15 descending diagonals using integers $\{-7, -6, \ldots, 6, 7\}$ and $\{0, 1, 2, \ldots, 14\}$ respectively.

Since a row cannot have more than one queen, and there are 8 rows and 8 queens, each row must have exactly one queen. Similarly, each column must have exactly one queen. So a solution placement of the 8 queens can simply be specified by mentioning for each row the column number where that row's queen is placed. Specifically, the solution is a permutation of all the integers from $\mathbb{Z}_8$, $(a_0, a_1, a_2, \ldots, a_7)$, where a_r is the column number for the queen on row r.

Such permutation must also reflect the constraint that two queens cannot share a diagonal. Note that for any r, values $(r - a_r)$ and $(r + a_r)$ would identify respectively the ascending and descending diagonal occupied by the queen on row r. We can in fact express the 8-Queens Problem as a simple abstract problem which deals only with integers:

Find a permutation of all the integers from $\mathbb{Z}_8$, $(a_0, a_1, a_2, \ldots, a_7)$, such that for all r, $(r - a_r)$ values are distinct, and $(r + a_r)$ values are distinct too.

With this abstract version, we no longer need to deal with the objects (chessboard, queens, their associated geometry) and rules of the original problem; we are left only with the necessary mathematical conditions underlying the problem.

In the rest of this discussion, we will be solving this abstract problem, the solution to which easily maps to the 8-Queens Problem. The constraint of $(r - a_r)$ and $(r + a_r)$ distinctness will be referred as constraint C.

An Algorithm

A trivial method is to generate all the 8! permutations of $\mathbb{Z}_8$ (using some permutation generation algorithm) and for each permutation check whether it satisfies the constraint C.

But a simple observation can help us optimize. Any permutation of m integers from $\mathbb{Z}_8$ ($m < 8$) is actually a prefix of some complete permutations of the 8 integers. Let us refer any such permutation as "permutation-prefix". If a permutation-prefix does not satisfy constraint C for its m elements, then any permutation of the 8 integers starting with this permutation-prefix will not satisfy the constraint too; there are $(8 - m)!$ such permutations. It means, we will not need to check these $(8 - m)!$ permutations individually, and can reject them as a whole.

To utilize this observation, we will generate permutations in such a way that all permutations starting with a common permutation-prefix (of any length) get generated together as a separable subtask. So, if a permutation-prefix is found to violate the constraint, we can easily skip the generation (and checking) of all permutations sharing this prefix.

One such way of generating permutations has been shown below (for integers in $\mathbb{Z}_8$), implemented in C. It need not be an efficient permutation generation algorithm, but is simple to understand. *permute*(s) is supposed to generate all permutations of the integers not already placed till index $(s - 1)$, over indices s to $N - 1$. *print*() prints the complete permutation from array a.

```
#define N 8

/* below array declarations assume N = 8 */

int a[8];

/* boolean array to track the integers (0 to 7) which have
   already been placed in the permutation being constructed */
char placed[] = {0, 0, 0, 0, 0, 0, 0, 0};

void permute(int s)
{
  int i;

  if(s == N)
  {
    print();
    return;
  }
  for(i = 0; i < N; i++)
  {
    if(!placed[i])
    {
      a[s] = i;
      placed[i] = 1;

      permute(s+1);

      placed[i] = 0;
    }
  }
}

void print()
{
  int i;

  for(i = 0; i < N; i++)
    printf("%d ", a[i]);

  printf("\n");
}
```

We will now modify this permutation generation method to solve the 8-Queens Problem. As mentioned earlier, if a permutation-prefix violates the constraint, we will skip generating (and checking) all permutations sharing this prefix. This skipping becomes easy here because all such permutations would get generated under a single recursive call $\mathit{permute}(s+1)$.

To record the $(r - a_r)$ and $(r + a_r)$ values already taken by the current permutation-prefix, we will use two boolean arrays. The modified method is shown below with all added lines marked using `/* 8Q */`. Method $\mathit{init}()$ should be called first for doing initializations.

```
/* boolean arrays to track "r plus a[r]" (rpar) which ranges from
   0 to 14, and "r minus a[r]" (rmar) which ranges from -7 to 7.
   The values of (r+a[r]) and (r-a[r]+7) will be used to index
   into these arrays. */
char rpar[15], rmar[15];

/* can the permutation-prefix be extended by placing "e" at
   index "r", without violating constraint C ? */
#define prefix_extensible(r, e) (!rpar[r+e] && !rmar[r-e+7])

/* the permutation-prefix was extended by including index "r" */
#define prefix_extended(r) {rpar[r+a[r]] = 1; rmar[r-a[r]+7] = 1;}
```

```
/* the permutation-prefix was reduced by excluding index "r" */
#define prefix_reduced(r) {rpar[r+a[r]] = 0; rmar[r-a[r]+7] = 0;}

void init()
{
  memset(rpar, 0, 15*sizeof(char));
  memset(rmar, 0, 15*sizeof(char));
}

void permute_8queens(int s)
{
  int i;

  if(s == N)
  {
    print();
    return;
  }

  for(i = 0; i < N; i++)
  {
    if(!placed[i])
    {
      if(!prefix_extensible(s, i))                    /* 8Q */
        continue;                                     /* 8Q */

      a[s] = i;
      placed[i] = 1;
      prefix_extended(s);                             /* 8Q */

      permute_8queens(s+1);

      placed[i] = 0;
      prefix_reduced(s);                              /* 8Q */
    }
  }
}
```

For placing element at index s, this permutation generation algorithm searches for an element which is not already placed till index $(s-1)$. We will now see a more efficient algorithm to generate permutations.

A Swapping Algorithm

An earlier article titled *Generating Permutations with Recursion* [1] discusses a permutation generation algorithm by C. T. Fike which can be modified to solve the 8-Queens Problem (refer method *fike*() in its section “Fike’s Algorithm”). Instead of searching, it systematically locates the next element to place at index s.

The modified method is shown below with all added lines marked using `/* 8Q */`. Unlike *permute*(), array a needs to be initialized in this case. For other initializations, the earlier shown *init*() method should be used.

```
int a[] = {0, 1, 2, 3, 4, 5, 6, 7};

#define SWAP(i, j) {int t; t = a[i]; a[i] = a[j]; a[j] = t;}

void fike_8queens(int s)
{
  int i;

  if(s == N-1)
  {
    if(prefix_extensible(s, a[s]))                    /* 8Q */
      print();

    return;
  }

  for(i = 0; i < N-s; i++)
  {
    if(!prefix_extensible(s, a[s+i]))                 /* 8Q */
      continue;                                       /* 8Q */

    if(i > 0)
      SWAP(s, s+i);

    prefix_extended(s);                               /* 8Q */

    fike_8queens(s+1);

    prefix_reduced(s);                                /* 8Q */

    if(i > 0)
      SWAP(s, s+i);
  }
}
```

Notice how, adding a few steps to the two permutation generation methods (*permute*() and *fike*()), gave us methods to solve an initially dissimilar 8-Queens Problem.

For both these methods, we could skip the recursive call (for argument $(s+1)$) whenever required and could still uphold the correctness of the permutation generation logic with ease. We may be able to use other permutation generation algorithms too (e.g. from article [1]) for the 8-Queens Problem, but introduction of such skipping may demand other changes to ensure their correctness.

■

References

[1] Nitin Verma. *Generating Permutations with Recursion.* `https://mathsanew.com/articles/permutations_with_recursion.pdf`.