Unit 27: N Queens

Learning Objectives

After taking this unit, students should

- be familiar with using recursion to search and prune the solution space to a problem
- be familiar with the class N queens problem and its solution

Searching and Bracktacking

if current rank fails, then will go back 1 rank to change that input and try again

We now look at how recursion can help with solving problems that require *searching and backtracking*. A classical example for this is the n-queens problem, which can be stated as given a $n \times n$ chessboard, find a possible placement of n queens on the chessboard, such that the queens do not threaten each other. In other words, there is exactly one queen in each row, in each column, and each diagonal. The 8-queens puzzle was first published by Max Bezzel in 1848, with the first solution published Franz Nauck in 1850. The generalized n-queens problems were introduced later. It is known that there is no solution for $n = 2$ and $n = 3$, but a solution exists for $n > 3$.

If we visualize the chessboard as a 2D array, with `#` as the position of a queen, and `.` as an empty position on the board, then a solution to the 4-queens problem looks like this:

```
.#..
...#
#...
..#.
```

Recursive Formulation

Let's see how we can formulate the problem recursively. The first step is to simplify the problem to the most trivial case where we can solve it. It is tempting to say that a simpler version of the problem is an $n - 1$-queens problem, and so the most trivial case is 1-queen. While the solution to 1-queen is trivial, there is no solution to both 2-queen and 3-queen problems. Further, if we have found a solution to the $n - 1$-queens problem, extending it to a solution of the n-queens problem is not trivial.

Approach 1: Generate All Permutations

As a start, let's borrow the idea from Unit 26 and generate all permutations of the queens' positions. Let's label the columns `a`, `b`, `c`, .. etc. Since we know that there must be exactly one queen in each row, and one queen in each column, the positions of the queens can be represented as a string that is a permutation of `abcde..`. For instance, the solution of the 4-queen problem depicted above can be represented by `bdac`.

A simple algorithm is thus to generate all possible $n!$ permutations, and for each one, check if it is a valid placement. We already ensure that there is exactly one queen per row and one queen per column. It remains to check that the queens do not threaten each other diagonally.

Let's write a function that checks, given a string representation of the queen positions, whether there is any queen that threaten another or not:

```
bool has_a_queen_in_diagonal(char queens[], long len, long i) {
  char curr_col = queens[i];
  char left_col = curr_col - 1;
  char right_col = curr_col + 1;
  for (long row = i+1; row < len; row += 1) {
    if (queens[row] == left_col || queens[row] == right_col) {
      return true;
    }
    left_col -= 1;
    right_col += 1;
  }
  return false;
}

bool threaten_each_other_diagonally(char queens[], long len) {
  for (long i = 0; i < len; i += 1) {
    // for each queen in row i, check rows i+1 onwards,
    // on both left (-=1) and right (+=1) side, if there
    // is a queen in that column.
    if (has_a_queen_in_diagonal(queens, len, i)) {
      return true;
    }
  }
  return false;
}
```

Solving the n queens problem is then easy. The code below prints out all possible solutions to the n queens problem.

```
void nqueens(char queens[], long n, long row) {
  if (row == n-1) {
    if (!threaten_each_other_diagonally(queens, n)) {
      cs1010_println_string(queens);
    }
    return;
```

```
  }

  nqueens(queens, n, row + 1);
  for (long i = row + 1; i < n; i++) {
    swap(queens, row, i);
    nqueens(queens, n, row + 1);
    swap(queens, row, i);
  }
}
```

Approach 2: Pruning Impossible Solutions

One of the principles of writing efficient code is to avoid doing useless work. The code above, which tests all permutations, actually generates much useless work. Suppose the queens in the first two rows already threaten each other, then, there is no need to continue to generate all possible placements of queens for the remaining rows.

This concept is called *pruning* and is a key to speeding up many searching-based solutions. We want to prune away bad solutions as early as possible. This can be achieved easily, by checking if the queens already placed on the first k rows threaten each other.

```
void nqueens(char queens[], long n, long row) {
  if (row == n-1) {
    if (!threaten_each_other_diagonally(queens, n)) {
      cs1010_println_string(queens);
    }
    return;
  }

  if (!threaten_each_other_diagonally(queens, row)) {
    nqueens(queens, n, row + 1);
  }
  for (long i = row + 1; i < n; i++) {
    swap(queens, row, i);
    if (!threaten_each_other_diagonally(queens, row)) {
      nqueens(queens, n, row + 1);
    }
    swap(queens, row, i);
  }
}
```

Adding these two conditionals can speed up the code significantly.

Such an algorithm is also known as a *branch-and-bound* algorithm, where we attempt different possible solutions and prune away partial solutions that we know will never work. This is an important concept in artificial intelligence (AI) In fact, the n-queen problem is a typical example used in introductory AI courses for branch-and-bound search. Again, such solutions are more naturally expressed recursively.

Problem Set 27

Problem 27.1

In the code for Approach 2 above, we check if the queens placed on Rows 0 to `row` threaten each other, and call `nqueens` recursively only if these queens do not threaten each other. Identify the repetitive work being done in the calls `threaten_each_other_diagonally`, and suggest a way to remove the repetitive work.

Problem 27.2

Consider the code to generate all possible permutations of a string from Problem 26.1. Suppose that we restrict the permutations to those where the same character does not appear next to each other. Modify the solution to Problem 26.1 to prune away permutations where the same character appears more than once consecutively.

Appendix: Complete Code Written in Lecture

```c
#include "cs1010.h"
#include <stdbool.h>
#include <unistd.h>

void draw(char queens[], long n) {
  cs1010_clear_screen();
  for (long i = 0; i < n; i += 1) {
    for (char col = 'a'; col < 'a' + n; col += 1) {
      if (queens[i] == col) {
        putchar('#');
      } else {
        putchar('.');
      }
    }
    putchar('\n');
  }
  usleep(10000);
}

void swap(char a[], long i, long j) {
  char temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}

bool has_a_queen_in_diagonal(const char queens[], long len, long i) {
  char curr_col = queens[i];
  char left_col = curr_col - 1;
  char right_col = curr_col + 1;
  for (long row = i+1; row < len; row += 1) {
    if (queens[row] == left_col || queens[row] == right_col) {
      return true;
```

```
    }
    left_col -= 1;
    right_col += 1;
  }
  return false;
}

bool threaten_each_other_diagonally(char queens[], long len) {
  for (long i = 0; i < len; i += 1) {
    // for each queen in row i, check rows i+1 onwards,
    // on both left (-=1) and right (+=1) side, if there
    // is a queen in that column.
    if (has_a_queen_in_diagonal(queens, len, i)) {
      return true;
    }
  }
  return false;
}

bool nqueens(char queens[], long n, long row) {
  if (row == n-1) {
    draw(queens, n);
    if (!threaten_each_other_diagonally(queens, n)) {
      cs1010_println_string(queens);
      return true;
    }
    return false;
  }
  if (!threaten_each_other_diagonally(queens, row) &&
      nqueens(queens, n, row+1)) {
    return true;
  }
  for (long i = row+1; i < n; i+= 1) {
    swap(queens, row, i);
    if (!threaten_each_other_diagonally(queens, row) &&
        nqueens(queens, n, row+1)) {
      return true;
    }
    swap(queens, i, row);
  }
  return false;
}

int main()
{
  long n = cs1010_read_long();
  char *queens;
  queens = malloc((size_t)(n + 1) * sizeof(char));
  if (queens == NULL) {
    cs1010_println_string("error allocating memory");
    return -1;
  }
  char curr = 'a';
  for (long i = 0; i < n; i++) {
    queens[i] = curr;
    curr += 1;
  }
  queens[n] = '\0';
```

```
91
92      nqueens(queens, n, 0);
93
94      free(queens);
95  }
```