CS1010

Evan Tay | *evantay@comp.nus.edu.sg*
https://github.com/DigiPie/cs1010_tut_c09

evantay@comp.nus.edu.sg

Today’s plan

- Kahoot Quiz!
- Solutions for PS 27.1 / 27.2 (Self-study)
- Consultation

Kahoot!

NQUEENS

PS 27.1 | PS27.2

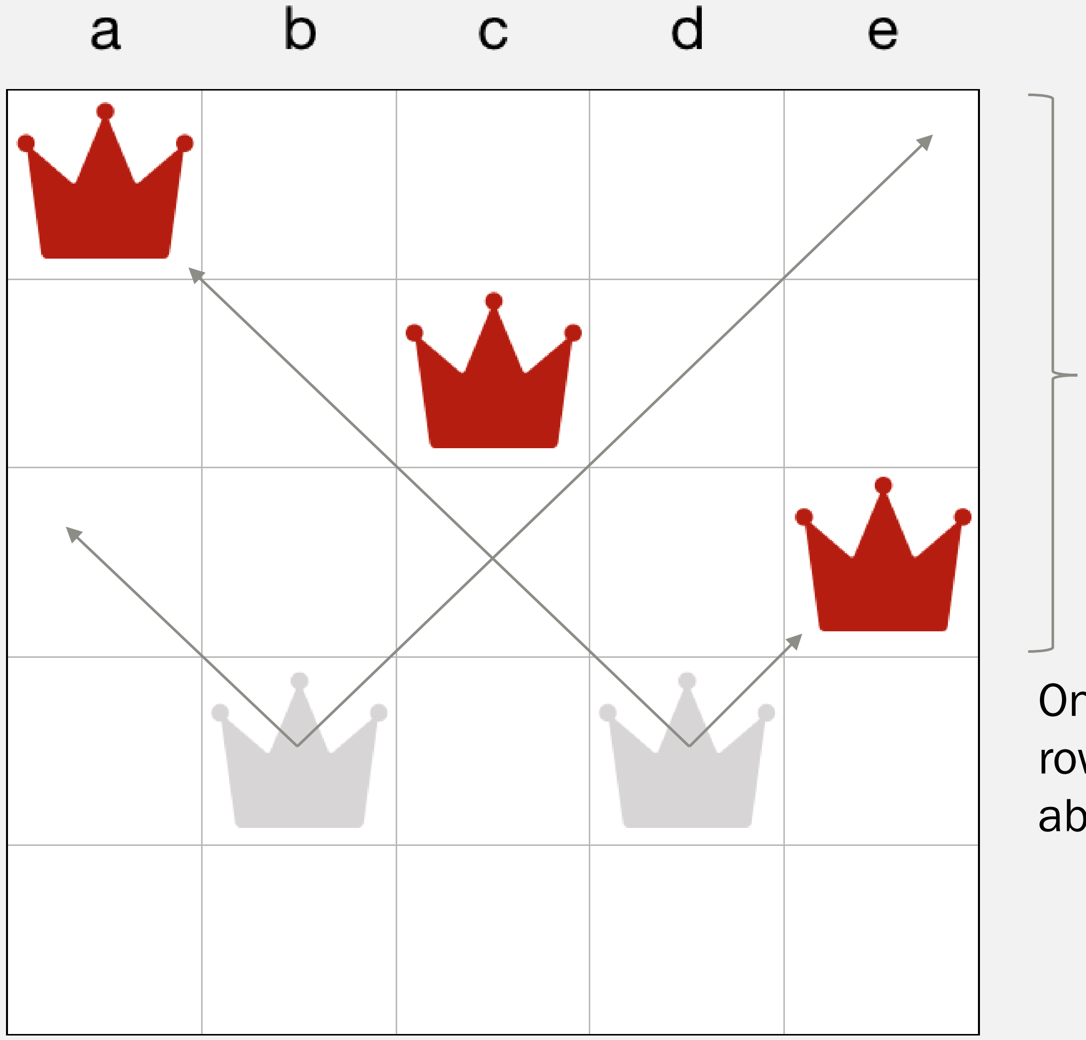
PS 27.1

```
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;
}
```

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.

evantay@comp.nus.edu.sg

PS 27.1 Solution



Already "safe."
No need to check again.

Only check the new row against the queens above.

evantay@comp.nus.edu.sg

PS 27.1 Solution

```
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;
}
```

When we recurse with nqueens(queens, n, row + 1); we already know that (!threaten_each_other_diagonally(queens, row)) is false. so queens on 0 to row-1 does not threaten each other. Now, we place a new queen on a new row, we do not need to repetitively check if the queens on row 0 to row-1 threaten each other. It suffices to check if the new queen threaten any of the queens in the previous rows!

PS 27.2

```
void permute(char a[], long len, long curr) {
 if (curr == len-1) {
   cs1010_println_string(a);
   return;
  }

  permute(a, len, curr + 1);
  for (long i = curr + 1; i < len; i += 1) {
   if (!appear_before(a, curr, i)) { // Line A
     swap(a, curr, i);
     permute(a, len, curr + 1);
     swap(a, i, curr);
   }
  }
}
```

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.

PS 27.2 Solution

- The key is to check if every "first character" of the permutation is the same with the previous character, if so, we can just prune the "search" process.
  - *Restrict the permutations to those where the same character does not appear next to each other.*
    - Prune away generation of permutations where the same characters appear next to each other.

PS 27.2 Solution

```
void permute(char a[], long len, long curr) {
  if (curr == len-1) {
    cs1010_println_string(a);
    return;
  }

  permute(a, len, curr + 1);
  for (long i = curr + 1; i < len; i += 1) {
    if (!appear_before(a, curr, i)) { // Line A
      swap(a, curr, i);
      permute(a, len, curr + 1);
      swap(a, i, curr);
    }
  }
}
```

```
void permute(char a[], long len, long curr) {
  if (curr == len-1) {
    if (a[curr] != a[curr-1]) {
      cs1010_println_string(a);
    }
    return;
  }
  if (a[curr] != a[curr-1]) {
    permute(a, len, curr + 1);
  }
  for (long i = curr + 1; i < len; i += 1) {
    if (!appear_before(a, curr, i) && a[i] != a[curr-1]) {
      swap(a, curr, i);
      permute(a, len, curr + 1);
      swap(a, i, curr);
    }
  }
}
```

THE END

Evan Tay | *evantay@comp.nus.edu.sg*

https://github.com/DigiPie/cs1010_tut_c09