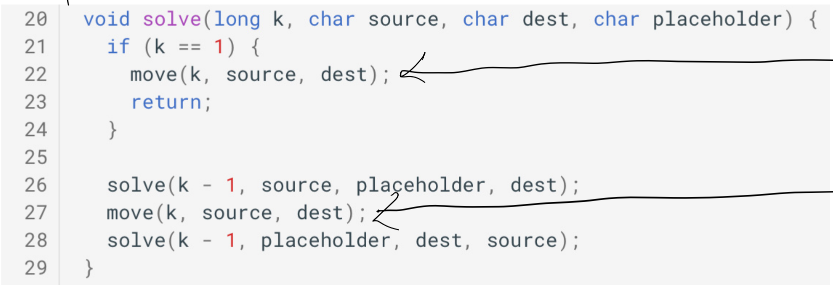
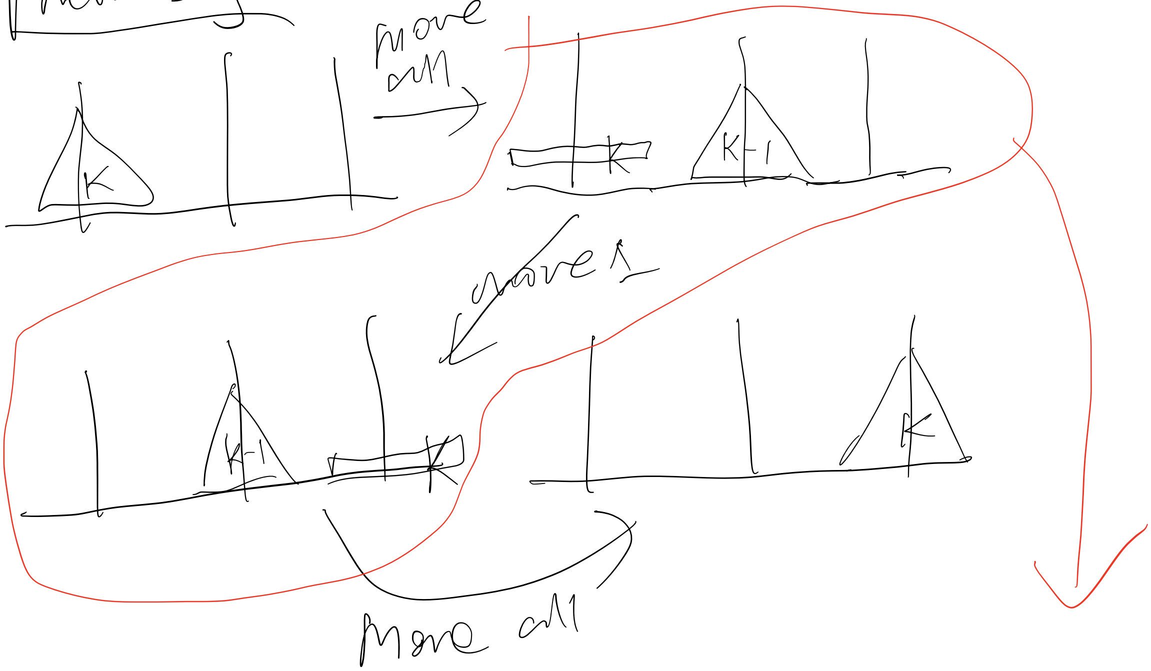
23.1



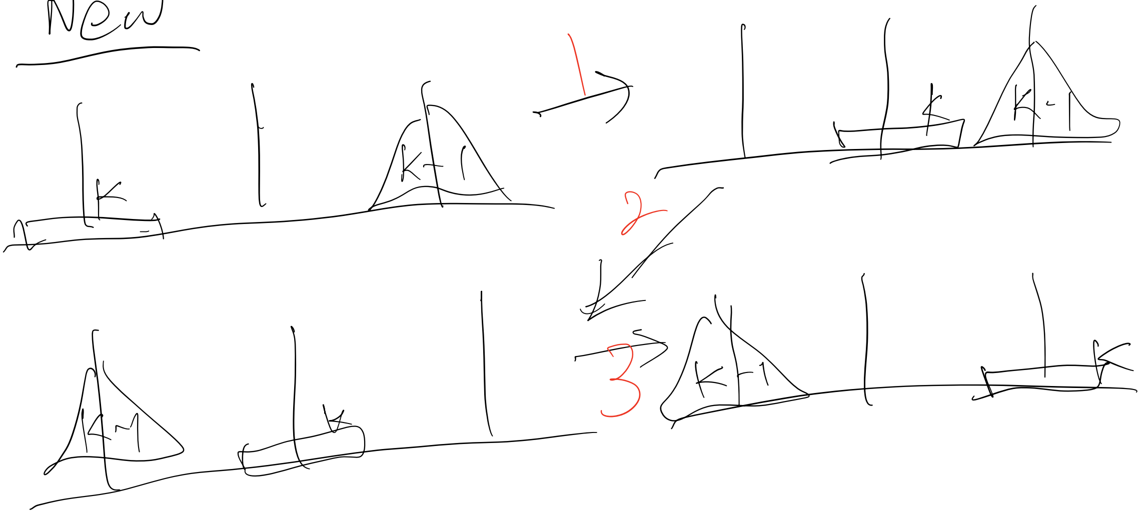
```
void solve(long k, char source, char dest, char placeholder) {
  if (k == 1) {
    move(k, source, dest);
    return;
  }

  solve(k - 1, source, placeholder, dest);
  move(k, source, dest);
  solve(k - 1, placeholder, dest, source);
}
```

Needs to be replaced.

Previously



New



```
void solve(long k, char source, char dest, char placeholder) {
  if (k == 1) {
    move(k, source, placeholder);
    move(k, placeholder, dest);
    return;
  }
  solve(k - 1, source, dest, placeholder);
  move(k, source, placeholder);
  solve(k - 1, dest, source, placeholder);
  move(k, placeholder, dest);
  solve(k - 1, source, dest, placeholder);
}
```

A → B
B → C

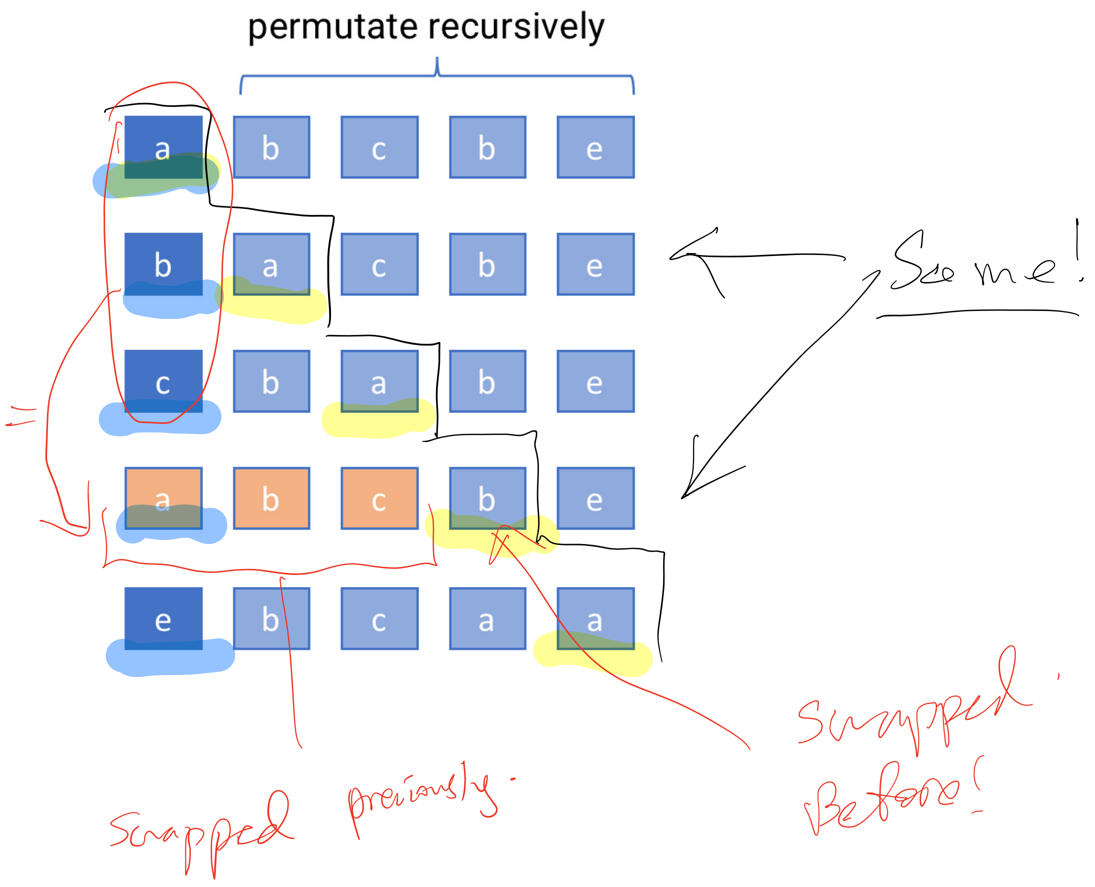
1
2
3

$$T(k) = \begin{cases} 3T(k-1) + 2, & \text{if } k \geq 1 \\ 2, & k = 1 \end{cases}$$

So, analysis follows ...

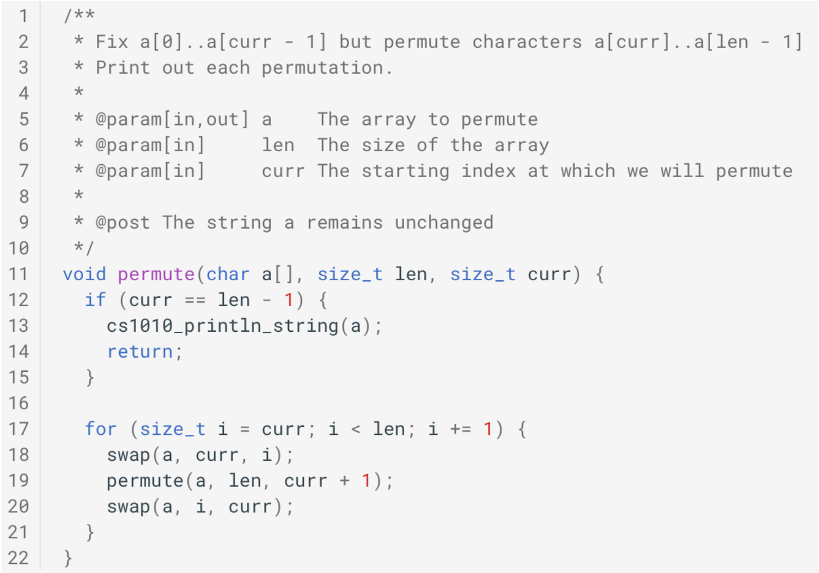
$O(3^k)$

24.1



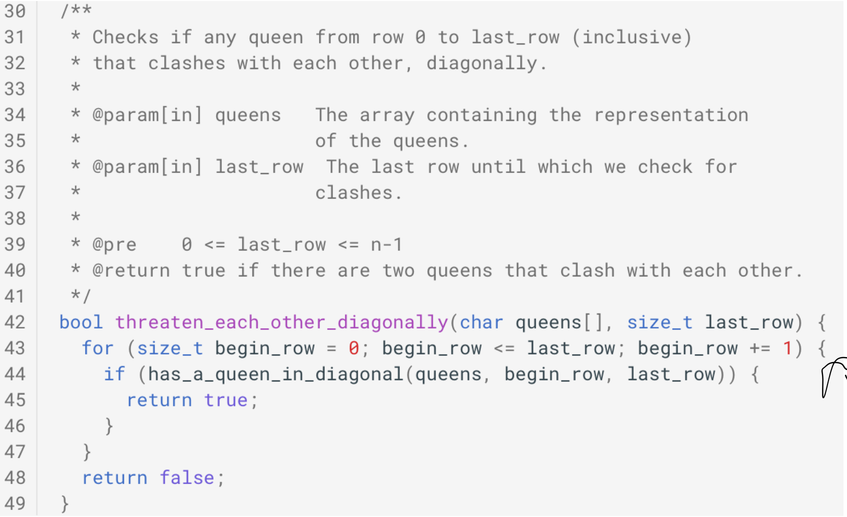
permutate recursively

a	b	c	b	e
b	a	c	b	e
c	b	a	b	e
a	b	c	b	e
e	b	c	a	a

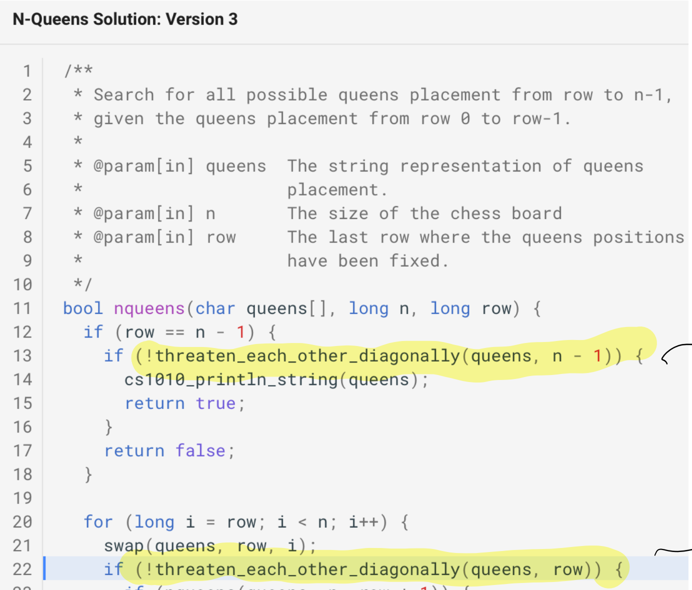


```
/**
 * Fix a[0]..a[curr - 1] but permute characters a[curr]..a[len - 1]
 * Print out each permutation.
 *
 * @param[in,out] a    The array to permute
 * @param[in]     len  The size of the array
 * @param[in]     curr The starting index at which we will permute
 *
 * @post The string a remains unchanged
 */
void permute(char a[], size_t len, size_t curr) {
  if (curr == len - 1) {
    cs1010_println_string(a);
    return;
  }

  for (size_t i = curr; i < len; i += 1) {
    swap(a, curr, i);
    permute(a, len, curr + 1);
    swap(a, i, curr);
  }
}
```

25.1

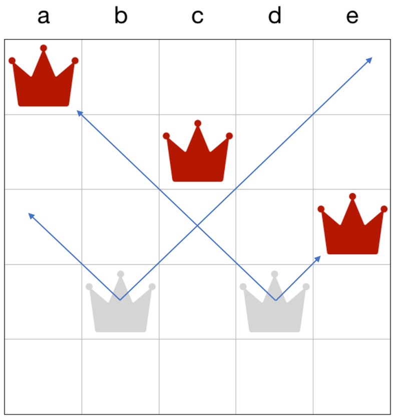


```
/**
 * Checks if any queen from row 0 to last_row (inclusive)
 * that clashes with each other, diagonally.
 *
 * @param[in] queens   The array containing the representation
 *                     of the queens.
 * @param[in] last_row  The last row until which we check for
 *                     clashes.
 *
 * @pre    0 <= last_row <= n-1
 * @return true if there are two queens that clash with each other.
 */
bool threaten_each_other_diagonally(char queens[], size_t last_row) {
  for (size_t begin_row = 0; begin_row <= last_row; begin_row += 1) {
    if (has_a_queen_in_diagonal(queens, begin_row, last_row)) {
      return true;
    }
  }
  return false;
}
```



N-Queens Solution: Version 3

```
/**
 * Search for all possible queens placement from row to n-1,
 * given the queens placement from row 0 to row-1.
 *
 * @param[in] queens  The string representation of queens
 *                    placement.
 * @param[in] n       The size of the chess board
 * @param[in] row     The last row where the queens positions
 *                    have been fixed.
 */
bool nqueens(char queens[], long n, long row) {
  if (row == n - 1) {
    if (!threaten_each_other_diagonally(queens, n - 1)) {
      cs1010_println_string(queens);
      return true;
    }
    return false;
  }

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



Already “safe.” No need to check again.

Only check the new row against the queens above.

25.2

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

Before swap'
a[curr]