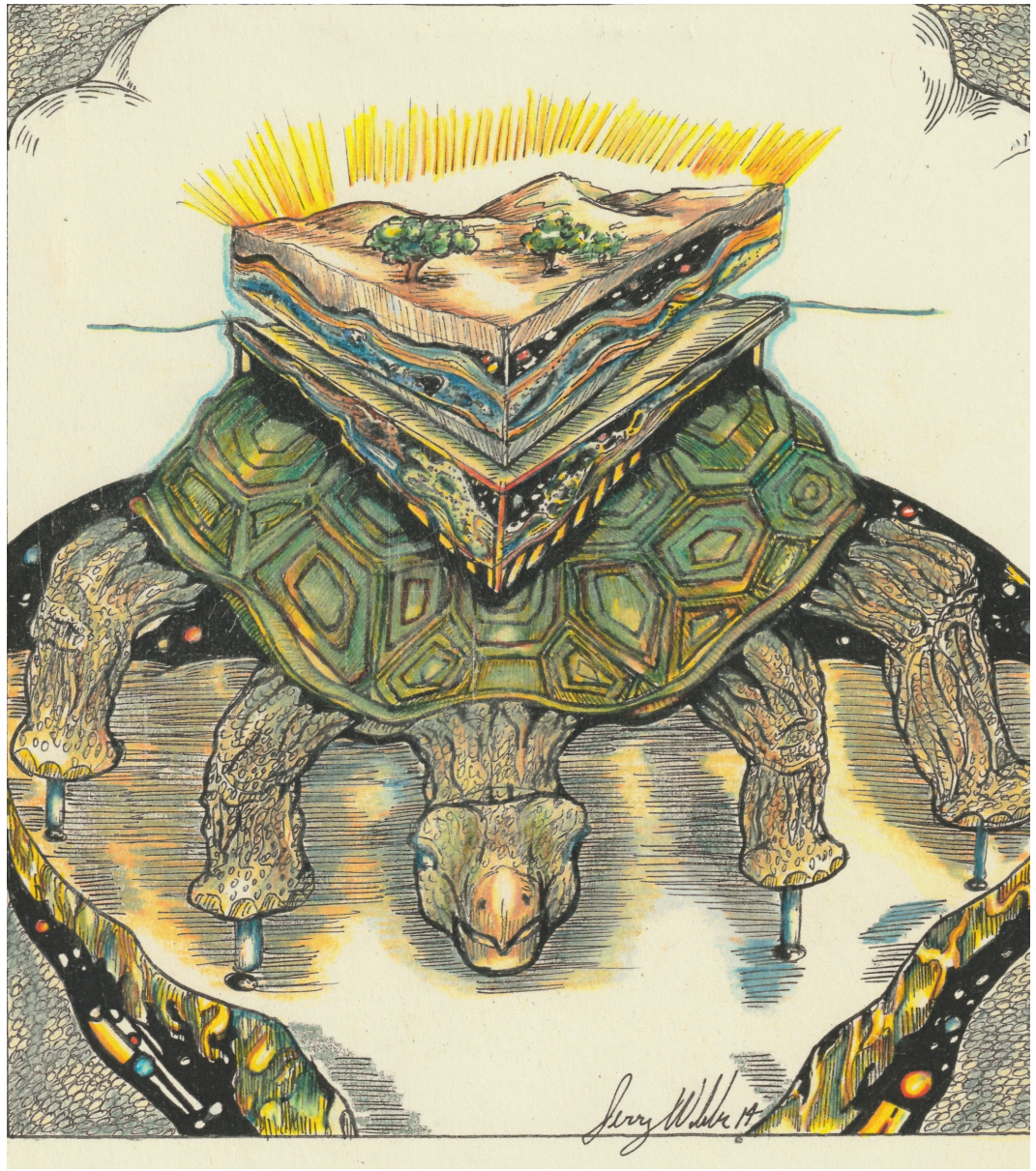
Recursion

Prof. Darrell Long

CSE 13S

"We can't stop here, this is bat country!"
—Hunter S. Thompson, *Fear and Loathing in Las Vegas*



A recursive function is defined in terms of *itself*.

In our favored version, an Eastern guru affirms that the earth is supported on the back of a tiger. When asked what supports the tiger, he says it stands upon an elephant; and when asked what supports the elephant he says it is a giant turtle. When asked, finally, what supports the giant turtle, he is briefly taken aback, but quickly replies "Ah, after that it is turtles all the way down."

—Antonin Scalia

Let's sum the first *k* natural numbers

- $\sum k = k + \sum(k - 1)$ and
- $\sum 1 = 1$, or
- We can loop through and add them up, or
- We can use Gauß's formula.
- The first of these is *recursive*, since it is defined in terms of itself.

```
int sum_1(int k) {
  if (k > 1) {
    return k + sum_1(k - 1);
  } else {
    return 1;
  }
}

int sum_2(int k) {
  int s = 1;
  for (int i = 2; i <= k; i += 1) {
    s += i;
  }
  return s;
}

int sum_3(int k) {
  return (k * (k + 1)) / 2;
}
```

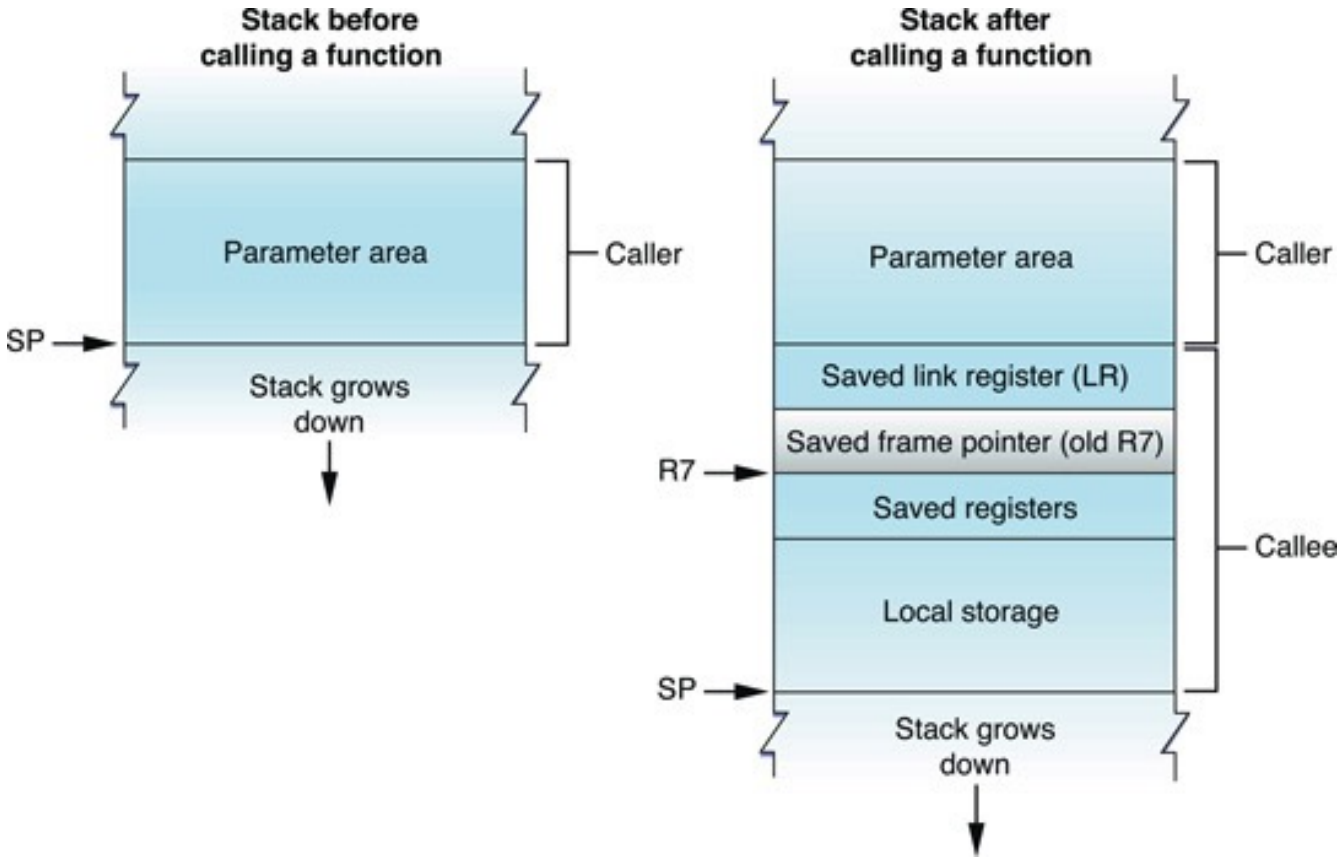
Fibonacci numbers

- In this case, both algorithms compute the same value.
- The first algorithm is recursive and runs in $O(2^n)$ time.
- The second algorithm is iterative and runs in $O(n)$ time.

```
int fib_1(int k) {
  if (k == 0 || k == 1) {
    return k;
  } else {
    return fib_1(k - 1) + fib_1(k - 2);
  }
}

int fib_2(int k) {
  int a = 0, b = 1, s = 0;
  if (k < 2) {
    return k;
  }
  for (int i = 2; i <= k; i += 1) {
    s = a + b;
    a = b;
    b = s;
  }
  return s;
}
```

Are recursive algorithms *inherently* inefficient?

- The short answer is *no*…
- The real answer is more subtle:
- A function call requires creating a *stack frame*, and that takes time and space.
- All tail recursive functions can be rewritten as iteration (often by the compiler).



So, when do I use recursion?

- The answer is simple: *when it makes sense.*
- Use recursion *when it is natural* for you to express your algorithm recursively.
- Do not use it when it does not make the code clearer!


```
int list[] = { 9, 3, 2, 7, 5, 1, 0, 4, 6, 8 };

int min_1(int *l, int k) {
  if (k == 1) {
    return *l;
  } else {
    int m = *l++;
    int n = min_1(l, k - 1);
    return m < n ? m : n;
  }
}

int min_2(int *l, int k) {
  int m = *l++;
  for (int i = 1; i < k; i += 1) {
    if (*l < m) {
      m = *l++;
    } else {
      l += 1;
    }
  }
```

Binary Search

- We can search an ordered array in $O(\log(n))$ time.

- If the list is *empty*, then it is not there.
- If the key is smaller, look in the left half;
- If it is larger then look in the right half.

```
int search(int key, int a[], int left, int right) {
  if (left > right) {
    return -1; // Not found
  } else {
    int middle = (left + right) / 2;
    if (key < a[middle]) {
      return search(key, a, left, middle - 1);
    } else if (k > a[middle]) {
      return search(key, a, middle + 1, right);
    } else {
      return middle;
    }
  }
}
```

You can also do this iteratively!

How fast can we search?

- Binary search is as fast as we can go.
- What about slicing it into three parts? Or 10?
- It's still $O(\log(n))$, just with a different constant.

- Remember: binary search requires that the array be *sorted*.

String Table

- Suppose that I want to efficiently store strings so that I only have a single copy of each one.
- Why would I want to do that?
- Compilers have to keep track of a lot of strings (variable names, for example).

```
#ifndef _STR_TREE_
#define _STR_TREE_

#include <strings.h>

typedef struct str_tree {
  char *entry;
  struct str_tree *left, *right;
} str_tree;

extern str_tree *str_root;

extern char *str_find(char *);
extern char *str_insert(char *);
#endif
```

`new_node()`

- Allocate a node,
- Set the children to NULL,
- Make room for the string,
- Copy the string.

- Did he really just use `strcpy()`?

```
static str_tree *new_node(char *key) {
  str_tree *node;

  node = (str_tree *)malloc(sizeof(str_tree));

  node->left  = (str_tree *)0; // New nodes have no
  node->right = (str_tree *)0; // children.

  node->entry = (char *)malloc(strlen(key) + 1);

  strcpy(node->entry, key);

  return node;
}
```

Bad!

`str_find()`

- Start at the root.
- Are we done?
- What does `strcmp()` say?
- Go left?
- Go right?
- There it is!

- Oops, did not find it.

```
char *str_find(char *key) {
  str_tree *chase;

  int direction;

  chase = str_root;

  while (chase != (str_tree *)0) {

    direction = strcmp(key, chase->entry);

    if (direction < 0) {
      chase = chase->left;
    } else if (direction > 0) {
      chase = chase->right;
    } else {
      return chase->entry;
    }
  }

  return (char *)0;
}
```

A *recursive* alternative

- If we've reached the leaves, then it isn't there.

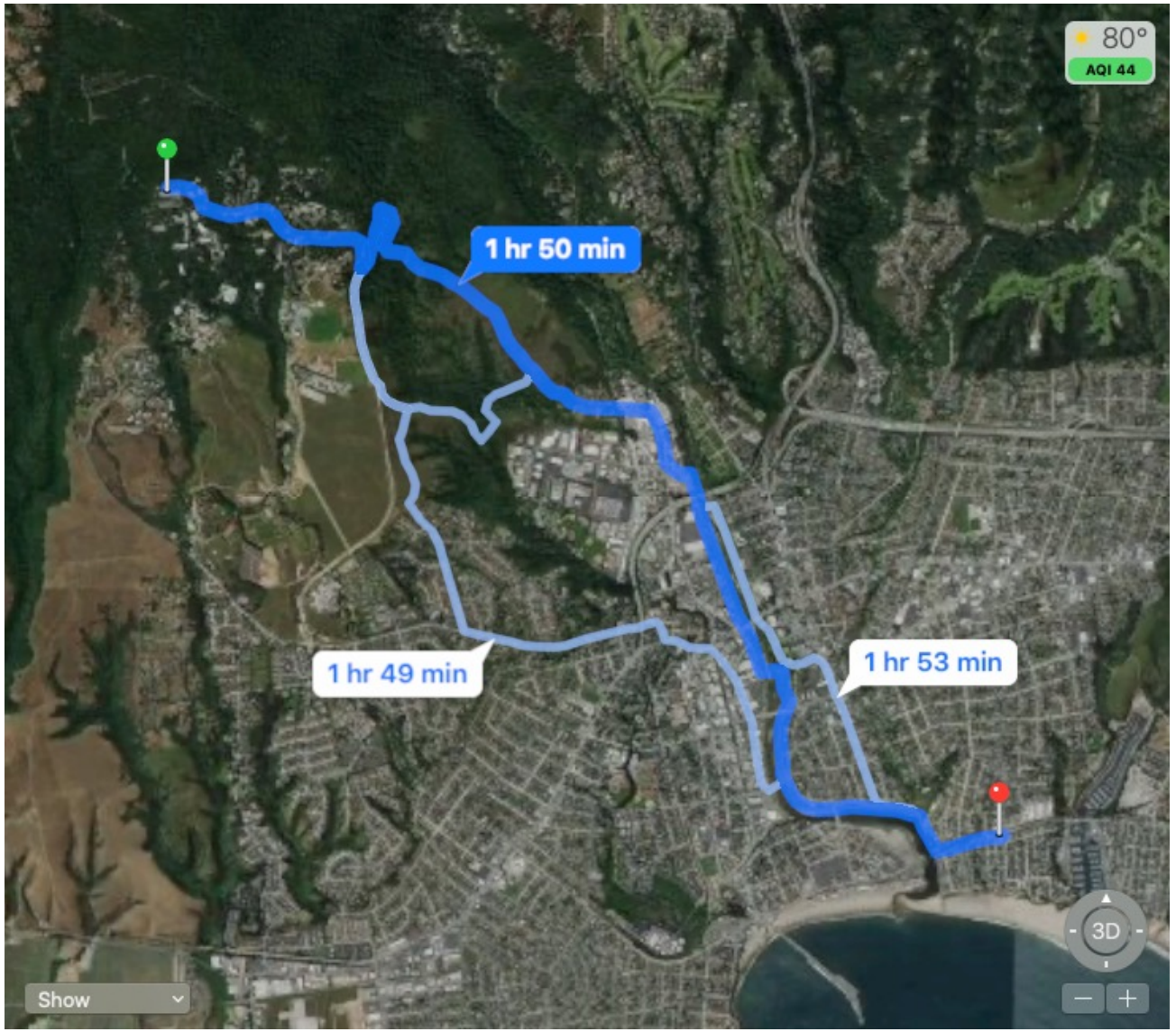
- What does `strcmp()` say?
- Recurse left?
- Recurse right?
- There it is!

```
char *str_find(str_tree *root, char *key) {

  int direction;

  if (root != (char *)0) {
    direction = strcmp(key, chase->entry);

    if (direction < 0) {
      return str_find(root->left, key);
    } else if (direction > 0) {
      return str_find(root->right, key);
    } else {
      return root->key;
    }
  }
  return (char *)0;
}
```

Recursion is natural for search

- We use recursion for searching by dividing the space up:
- Places *we have searched*, and
- Places *we have yet to search*.
- If we get stuck, we can back up and try a different path.

- For example, we want to find the shortest path from E2 to Betty Burgers.

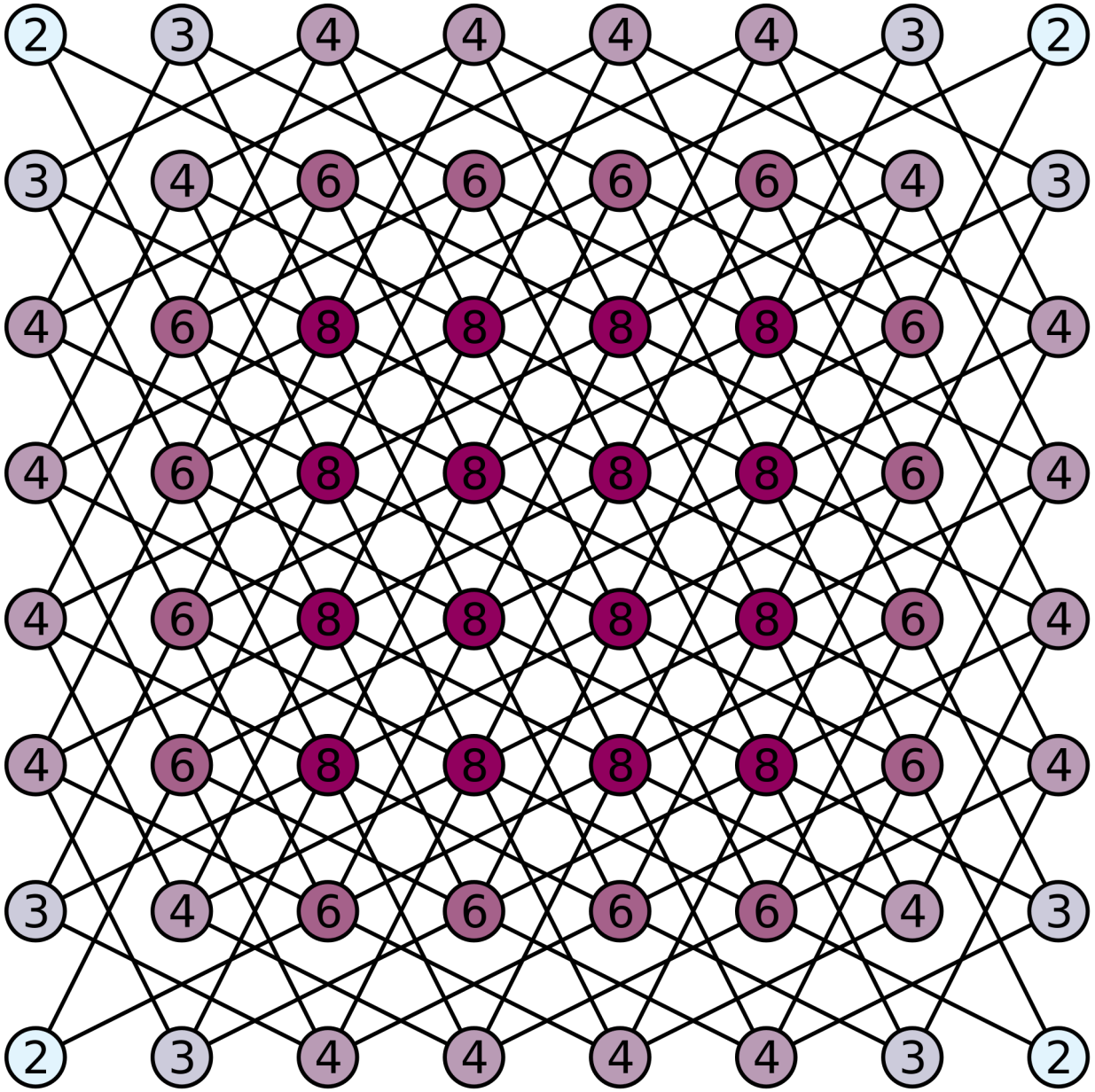


The Knight's Tour

```
bool attempt(uint32_t i, uint32_t x, uint32_t y) {
  uint32_t k = 0;
  bool quit;
  do {
    uint32_t u, v;
    count += 1;
    quit = false;
    u = x + a[k];
    v = y + b[k];
    if ((u < n && u >= 0) && (v < n && v >= 0)) { // On the board?
      if (chess_board[u][v] == 0) {               // Vacant?
        chess_board[u][v] = i;                    // Place knight
        if (i < n * n) {                          // Done?
          quit = attempt(i + 1, u, v);
          if (quit == false) {
            chess_board[u][v] = 0; // Back up and try again
          }
        } else {
          quit = true;
        }
      }
    }
    k += 1;
  } while (quit == false && k < 8);
  return quit;
}
```

Can a knight visit all of the squares on a chess board exactly once?

This is a form of a Hamiltonian path problem.



How many possible moves?

There are 26,534,728,821,064 directed tours on an 8 × 8 chess board.

Let's try it!

```
darrell@hilbert Examples % clang -DDEBUG=1 -o knight knight.c
darrell@hilbert Examples % ./knight -n 5
        1       6       15      10      21
        14      9       20      5       16
        19      2       7       22      11
        8       13      24      17      4
        25      18      3       12      23
Solution required 70624 tries.
darrell@hilbert Examples % ./knight -n 8
        1       60      39      34      31      18      9       64
        38      35      32      61      10      63      30      17
        59      2       37      40      33      28      19      8
        36      49      42      27      62      11      16      29
        43      58      3       50      41      24      7       20
        48      51      46      55      26      21      12      15
        57      44      53      4       23      14      25      6
        52      47      56      45      54      5       22      13
Solution required 66005601 tries.
darrell@hilbert Examples % ./knight -n 10
Try 1000000000
Try 2000000000 <-------- Perhaps not!
Try 3000000000
^C
```

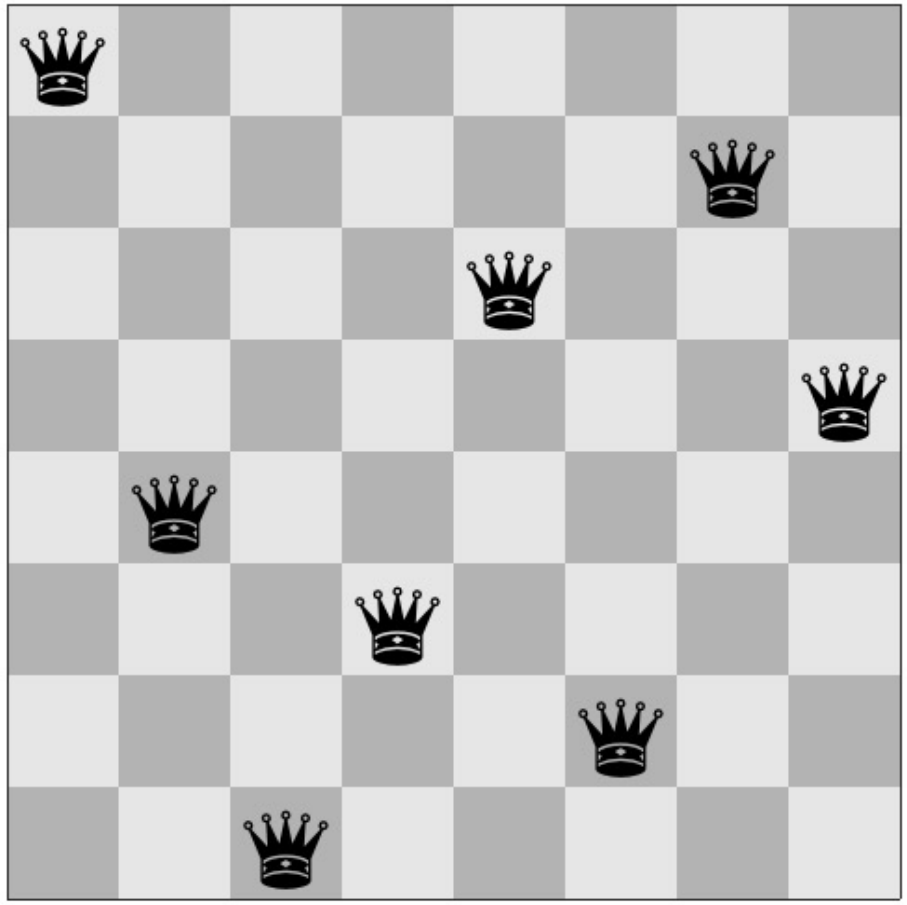
Eight Queens

Can we place eight queens on a chess board so that none threatens the others?

There are 92 solutions for an 8 × 8 chess board.

Larger boards have more solutions.

What is the expected run time of this problem?



Number of solutions for n × n, starting with n = 1:

1, 0, 0, 1, 2, 1, 6, 12, 46, 92, 341, 1787, 9233, 45752, 285053, 1846955, 11977939, 83263591, 621012754, 4878666808, 39333324973, 336376244042, 3029242658210, 28439272956934, 275986683743434, 2789712466510289, 29363495934315694

Print the Board

Solution due to Niklaus Wirth & Deepank Pant

```
void printSolution(int board[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      printf("%c ", board[i][j] ? 'Q' : '_');
    }
    printf("\n");
  }
}
```

Is this position safe?

```
bool isSafe(int board[N][N], int row, int col) {
  int i, j;
  // Rows
  for (i = 0; i < col; i++) {
    if (board[row][i]) { return false; }
  }
  // Left diagonal
  for (i = row, j = col; i >= 0 && j >= 0; i -= 1, j -= 1) {
    if (board[i][j]) { return false; }
  }
  // Right diagonal
  for (i = row, j = col; j >= 0 && i < N; i += 1, j -= 1) {
    if (board[i][j]) { return false; }
  }
  return true;
}
```

Search for the solution

```
bool solveNQP(int board[N][N], int col) {
  if (col >= N) { return true; }
  // Placing the queen in each row
  for (int i = 0; i < N; i++) {
    // Did that work?
    if (isSafe(board, i, col)) {
      // OK, do it!
      board[i][col] = 1;
      // Recurse to place the others
      if (solveNQP(board, col + 1)) {
        return true;
      }
      board[i][col] = 0; // Did not work back up!
    }
  }
  // No solution found!
  return false;
}
```

Did it work?

```
darrell@hilbert Examples % clang -o queens queens.c
darrell@hilbert Examples % ./queens
Q - - - - - - -
- - - - - - Q -
- - - - Q - - -
- - - - - - - Q
- Q - - - - - -
- - - Q - - - -
- - - - - Q - -
- - Q - - - - -
darrell@hilbert Examples %
```

What did we learn?

- Recursion is natural.
- It is good for search problems.
- It is not *inherently* inefficient.
- Like all tools and techniques, use it where it *makes sense*.