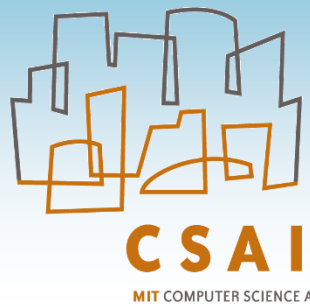


FROM

MODERN ALGORITHMS WORKSHOP

Parallel Algorithms

Prof. Charles E. Leiserson

Dr. Tao B. Schardl

September 19, 2018

Outline

- Introduction
- Cilk Model
- Detecting Nondeterminism
- What Is Parallelism?
- Scheduling Theory Primer
- *Lunch Break*
- Analysis of Parallel Loops
- Case Study: Matrix Multiplication
- Case Study: Jaccard Similarity
- Post-Moore Software

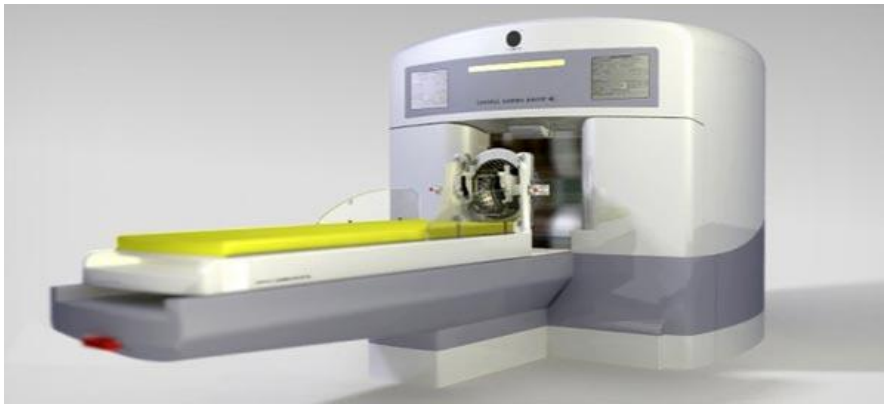
DETECTING NONDETERMINISM

Race Conditions

Race conditions are the bane of concurrency. Famous race bugs include the following:

- Therac-25 radiation therapy machine — killed 3 people and seriously injured many more.
- North American Blackout of 2003 — left 50 million people without power.

Race bugs are notoriously difficult to discover by conventional testing!

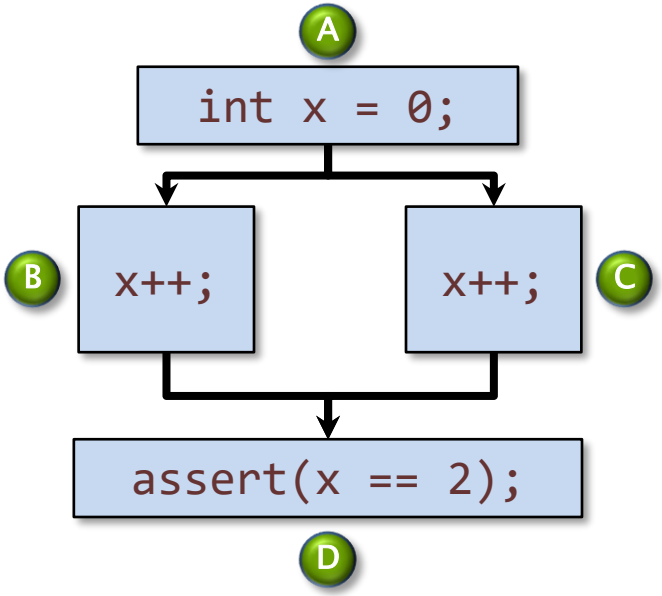


Determinacy Races

Definition. A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

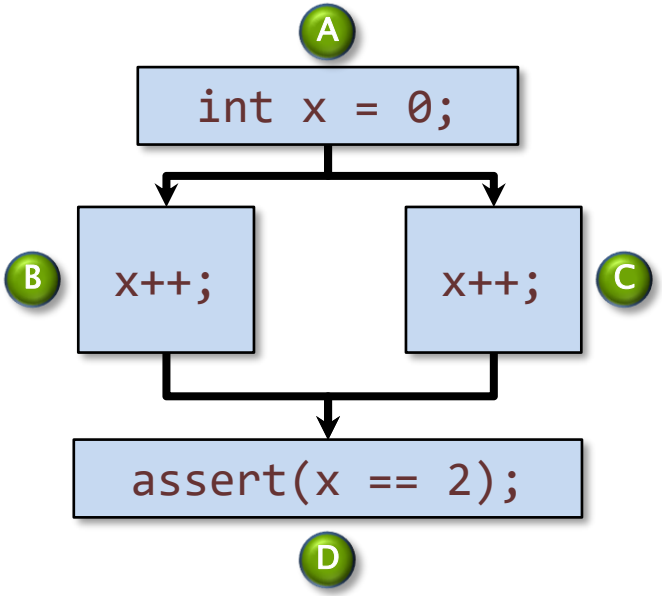
Example

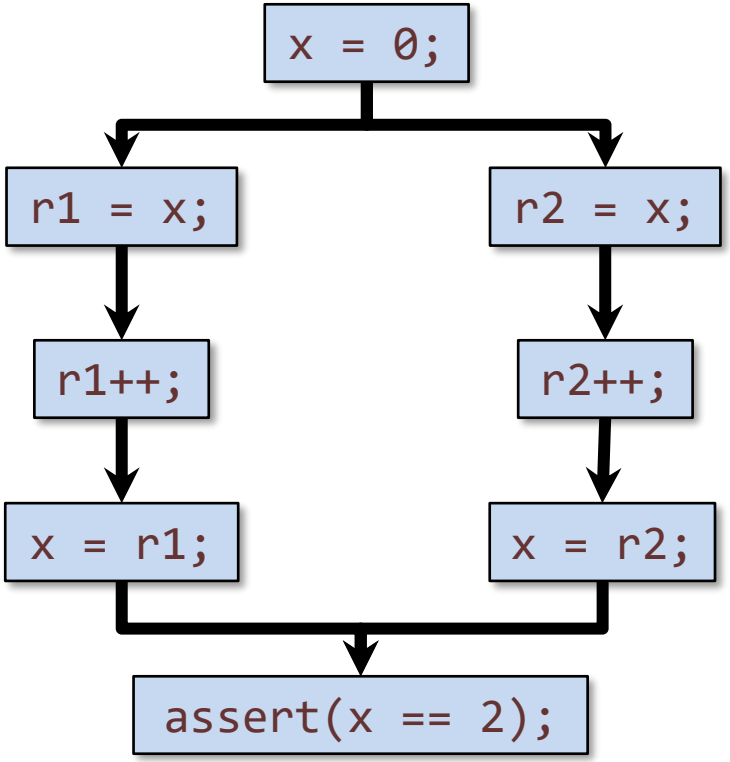
```
A  int x = 0;
   cilk_for (int i=0, i<2, ++i) {
B C    x++;
   }
D  assert(x == 2);
```



dependency graph

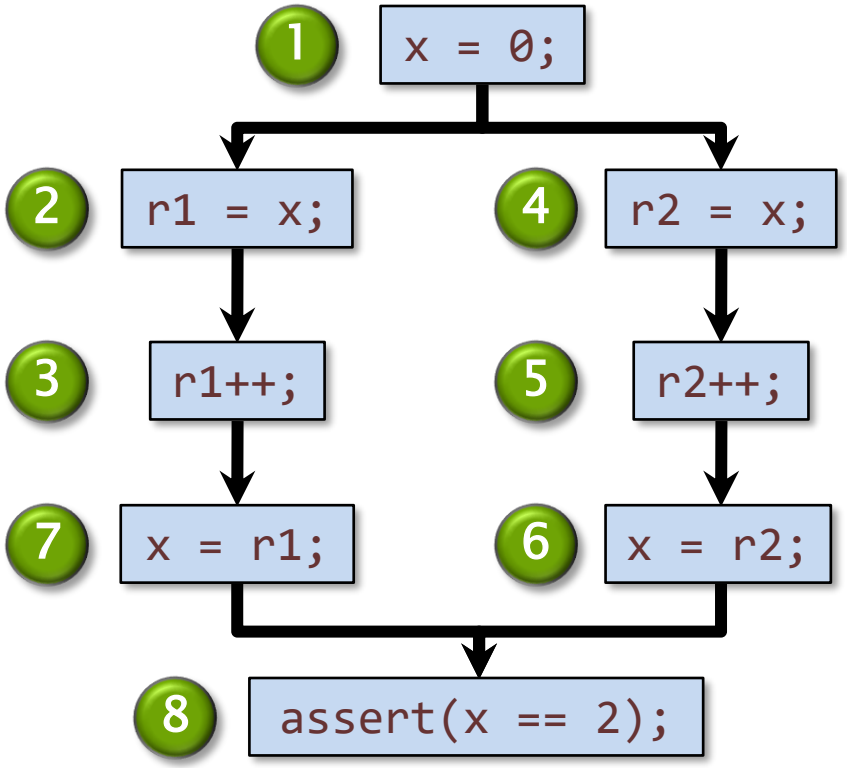
A Closer Look

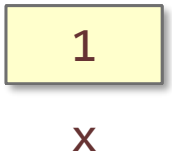


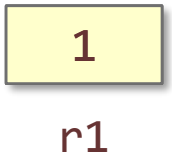


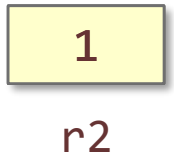
Race Bugs

Definition. A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.









Types of Races

Suppose that instruction A and instruction B both access a location x, and suppose that A∥B (A is parallel to B).

A	B	Race Type
read	read	none
read	write	read race
write	read	read race
write	write	write race

Two sections of code are independent if they have no determinacy races between them.

Determinacy Race ≠ Data Race

A determinacy race is **not** the same thing as a ***data race***, which occurs in the context of mutual-exclusion locking.

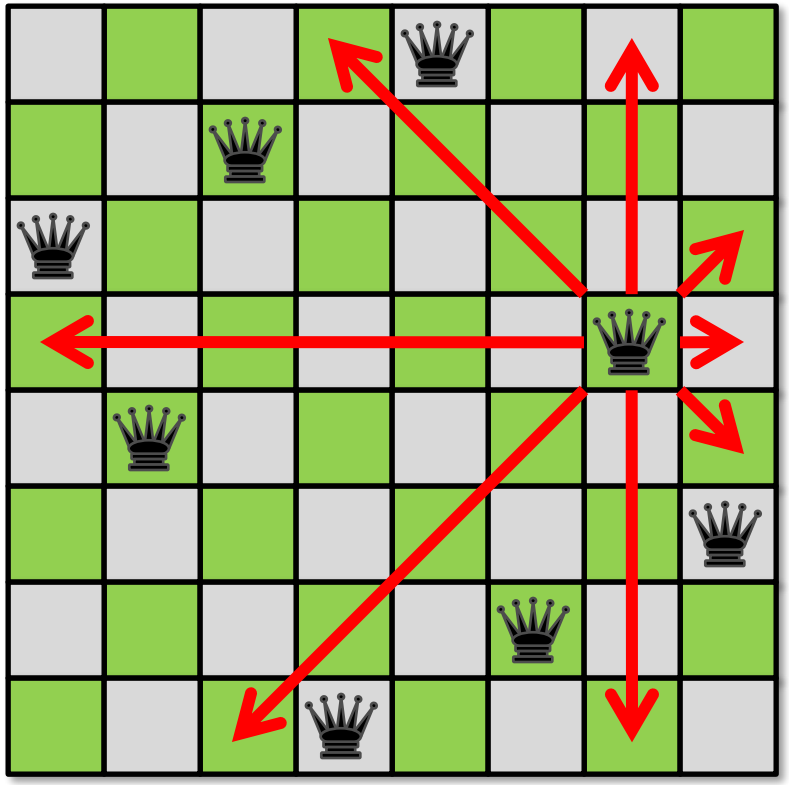
- If a program contains no determinacy races, then it contains no data races either.
- If a program contains no data races, it is **not necessarily** free of determinacy races.

EXAMPLE: QUEENS

Queens Problem

Problem

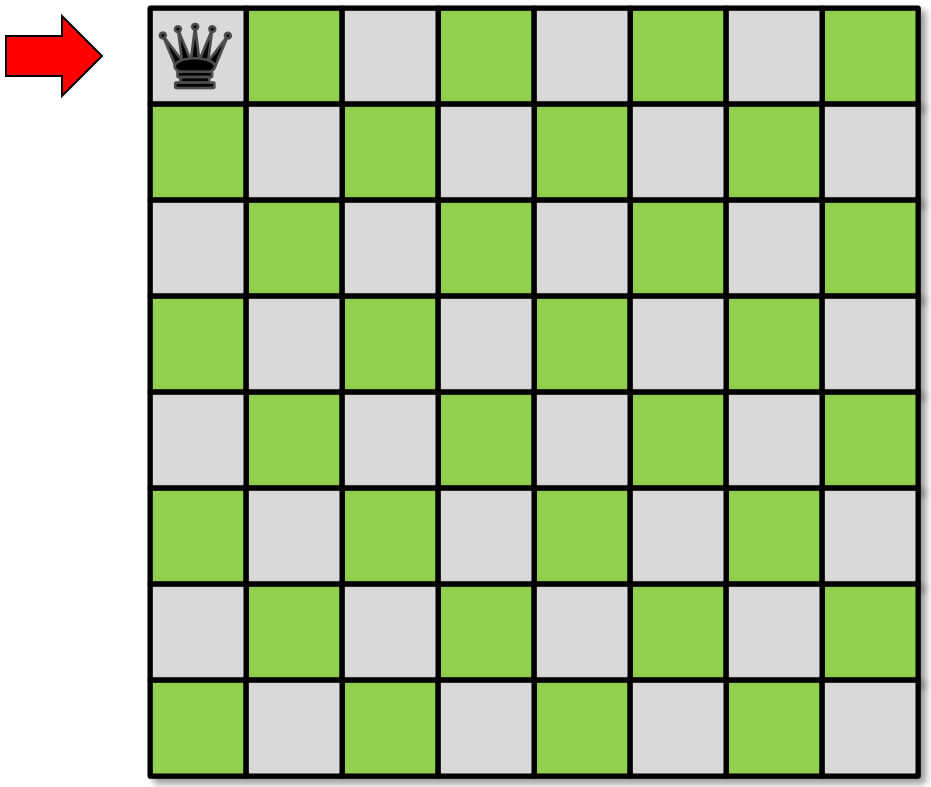
Place n queens on an $n \times n$ chessboard so that no queen attacks another, i.e., no two queens in any row, column, or diagonal. Count the number of possible solutions.



Backtracking Search

Strategy

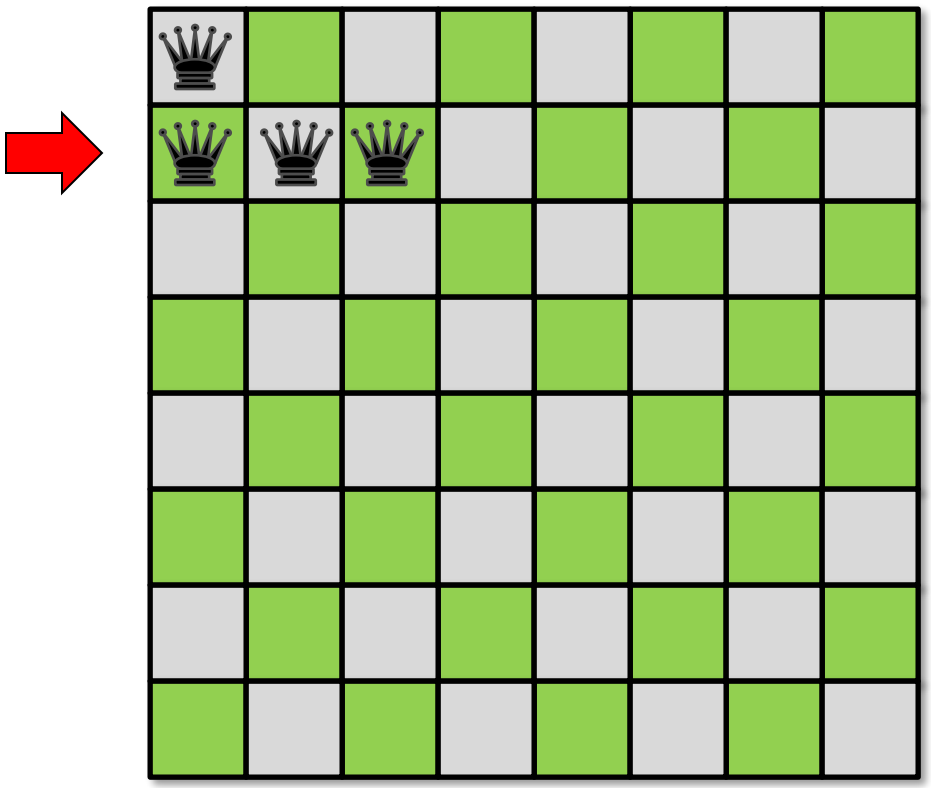
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

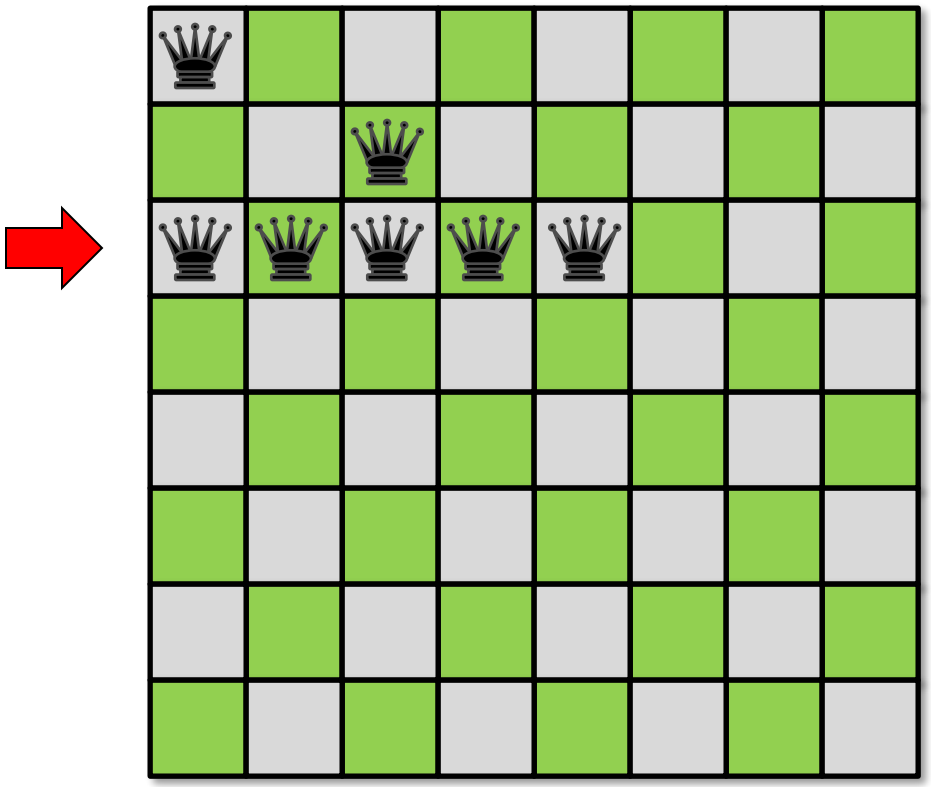
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

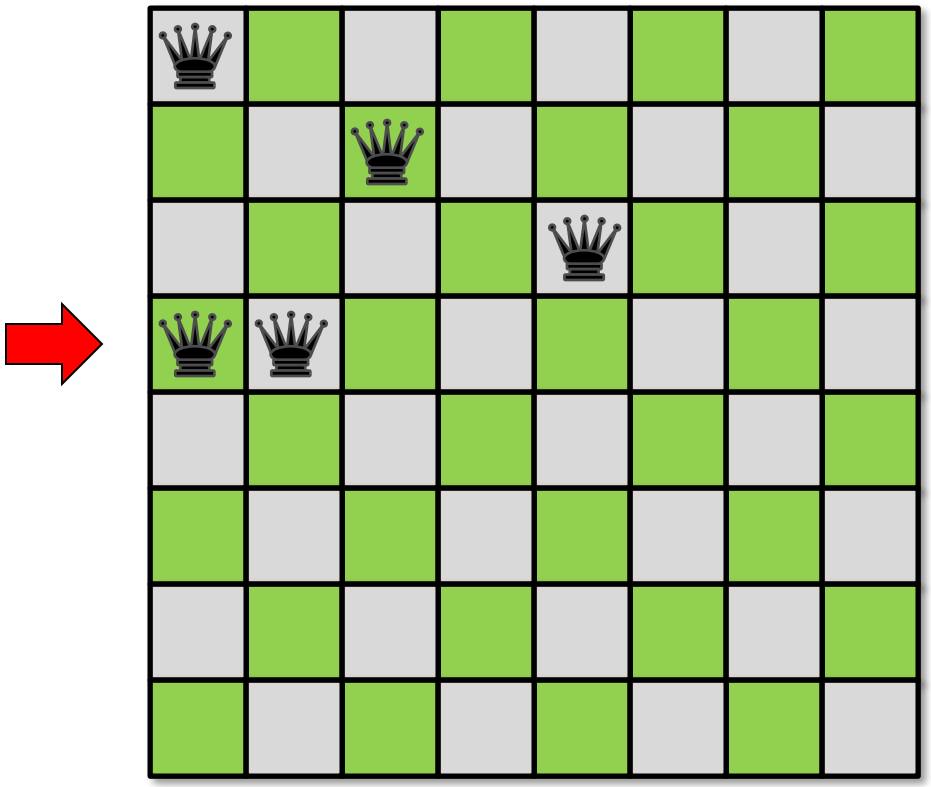
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

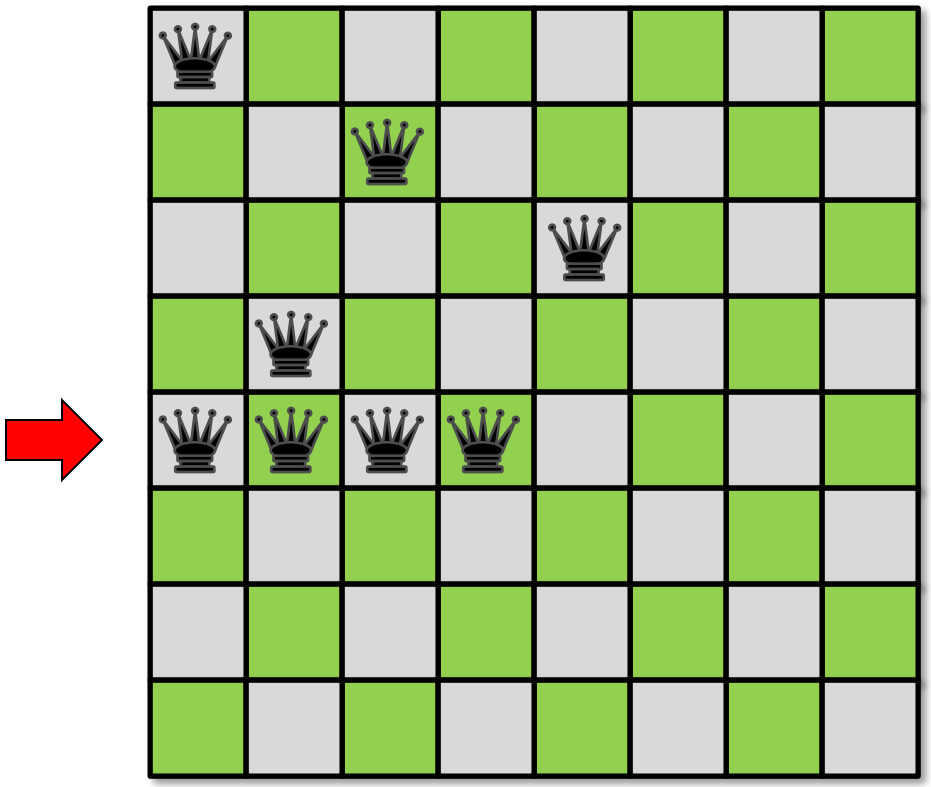
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

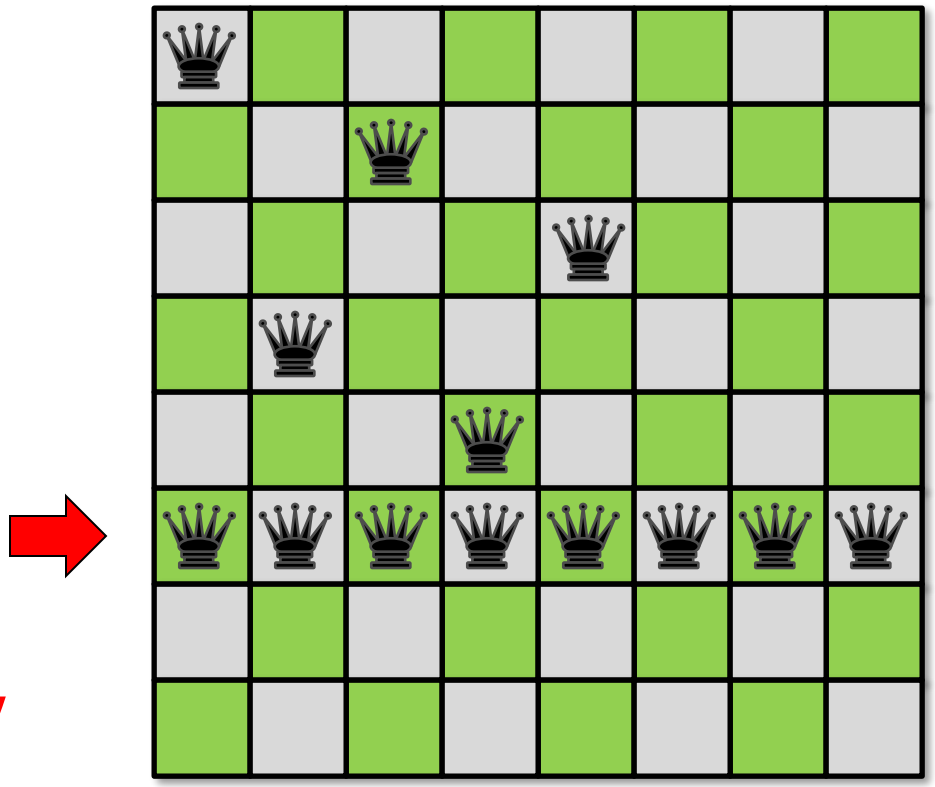
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

Try placing queens row by row. If you can't place a queen in a row, backtrack.

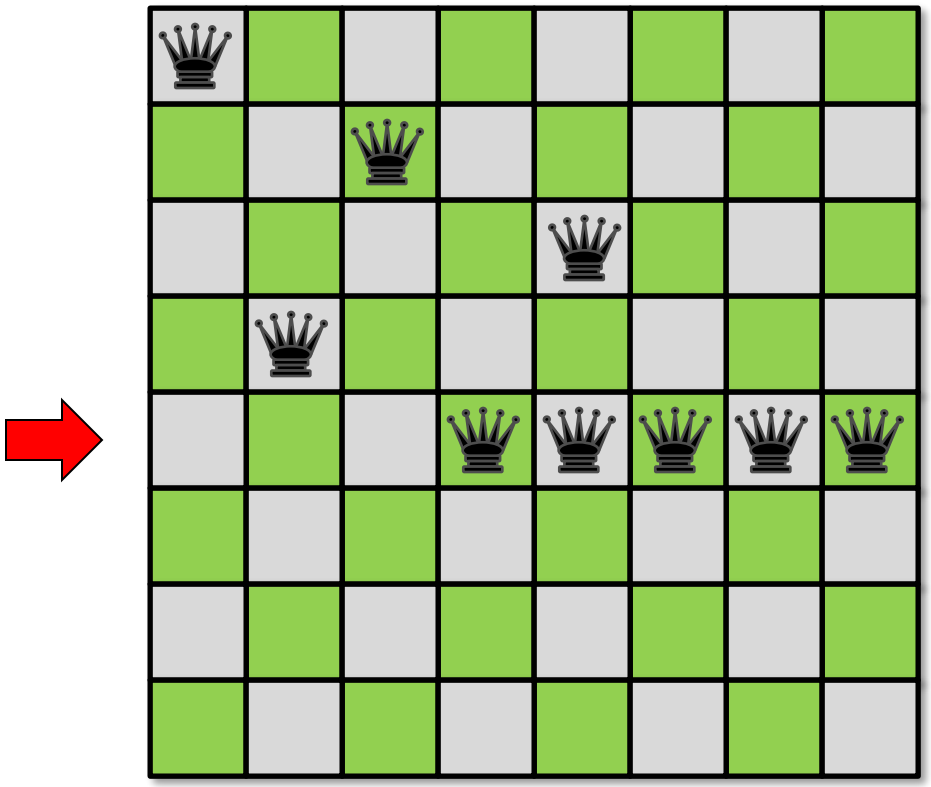


Backtrack!

Backtracking Search

Strategy

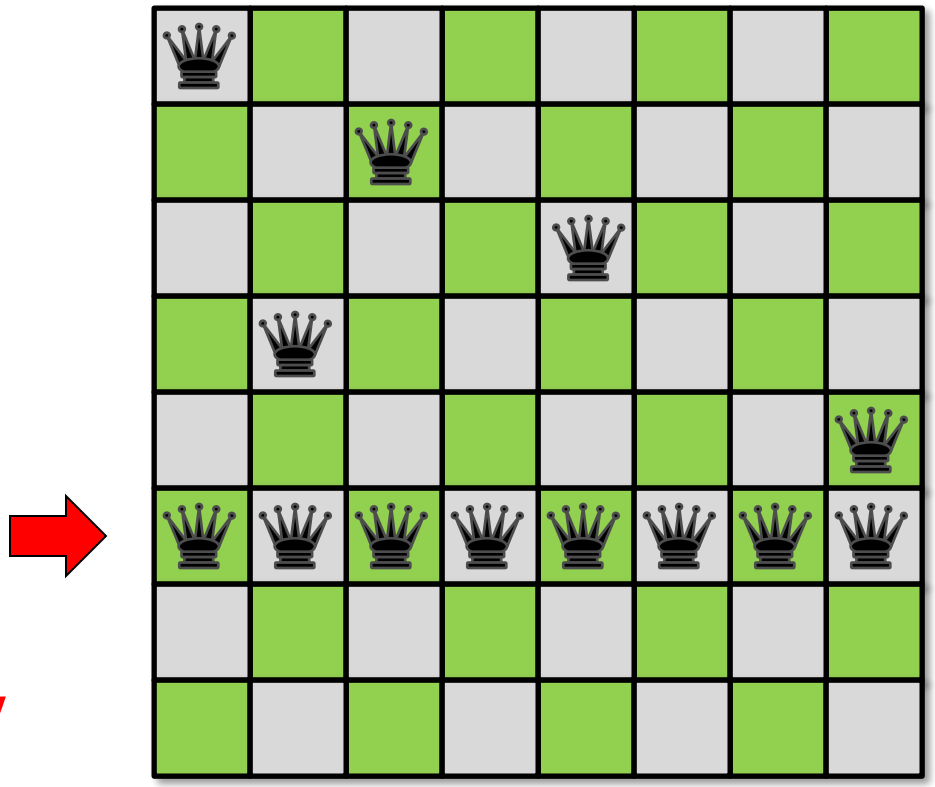
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

Try placing queens row by row. If you can't place a queen in a row, backtrack.

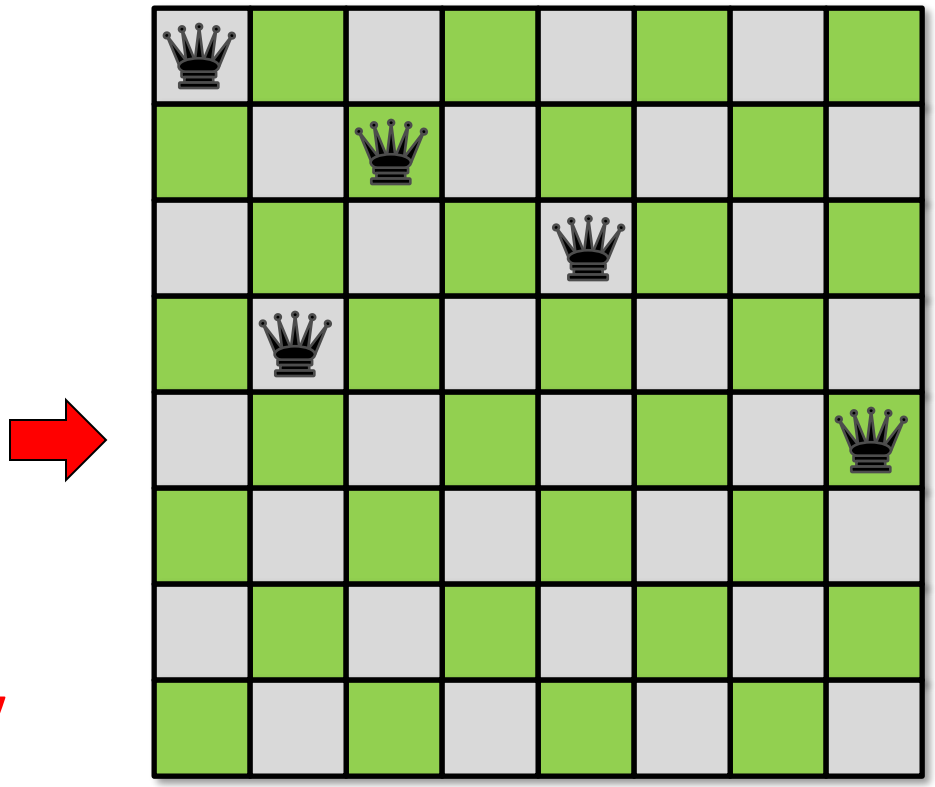


Backtrack!

Backtracking Search

Strategy

Try placing queens row by row. If you can't place a queen in a row, backtrack.

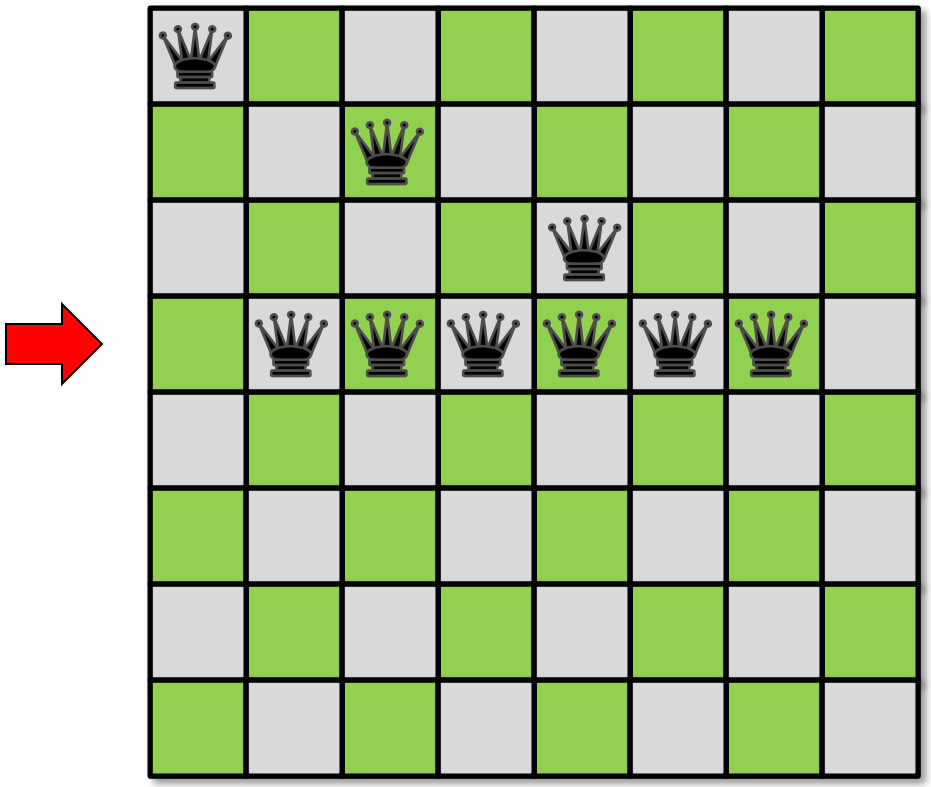


Backtrack!

Backtracking Search

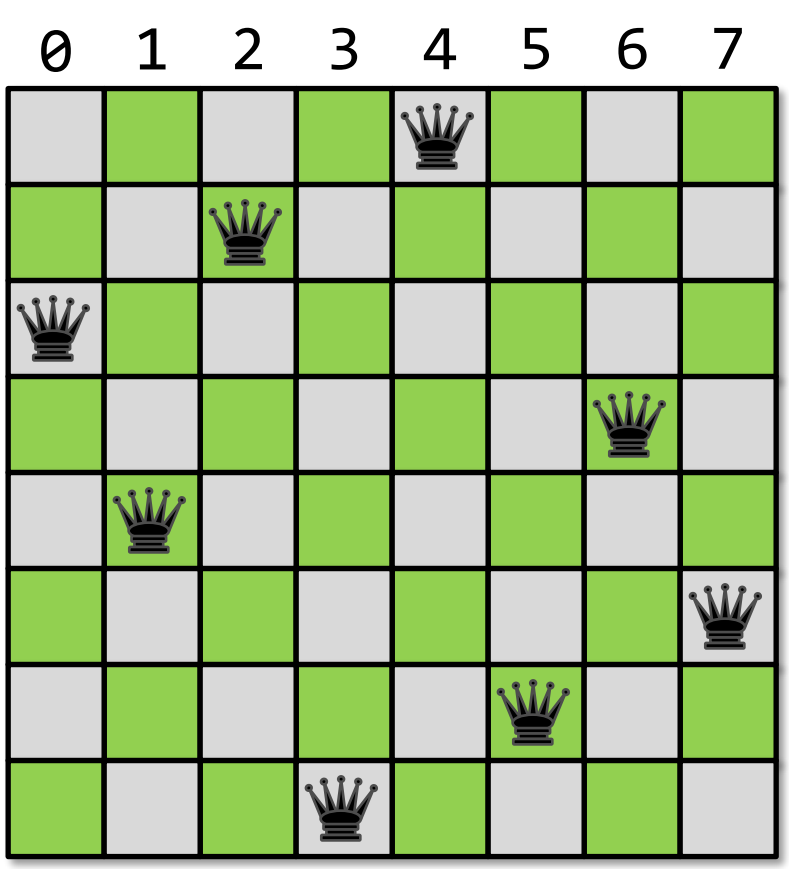
Strategy

Try placing queens row by row. If you can't place a queen in a row, backtrack.

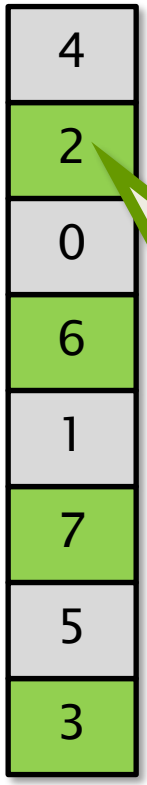


Board Representation

The board can be represented as an array of integers.



Representation



Column where the queen in this row is placed.

The board can be represented more efficiently, but that's another lecture.

Example: Queens Code (nqueens.c)

```
int nqueens(int n, int row, char *board_in) {
  if (n == row) return 1;  // Base case: Fully populated board
  char *board;
  int count[n];
  memset(count, 0, n * sizeof(int));

  // Make a local copy of the board
  board = (char *)alloca((row + 1) * sizeof(char));
  memcpy(board, board_in, row * sizeof(char));

  // Try each column for placing the queen.
  for (int col = 0; col < n; col++) {
    board[row] = col;
    // If this board is OK, continue search in parallel
    if (board_ok(row + 1, board))
      count[col] = cilk_spawn nqueens(n, row + 1, board);
  }
  cilk_sync;
  // Return the total number of solutions found.
  return total_solutions(count, n);
}
```

Where's the race?

HANDS-ON: THE CILKSAN DETERMINACY-RACE DETECTOR

Cilksan Race Detector

- The Cilksan-instrumented program is produced by compiling with the `-fsanitize=cilk` command-line compiler switch.
- If an ostensibly deterministic Cilk program run on a given input could possibly behave any differently than its serial projection, Cilksan guarantees to report and localize the offending race.
- Cilksan facilitates a regression-test methodology, where the programmer provides test inputs.
- Cilksan identifies filenames, lines, and variables involved in races, including stack traces.
- Ensure that all program files are instrumented, or you'll miss some bugs.
- Cilksan is your best friend.

Run Cilksan on Queens Code

1. Compile `nqueens.c` with Cilksan and run it to find the determinacy race:

```
$ cd nqueens
$ clang nqueens.c -o nqueens -fcilkplus -fsanitize=cilk -Og -g
$ ./nqueens 9
```

Build the program with Cilksan.

Include debug information.

2. Fix the race in `nqueens.c`. Use Cilksan to check your work.

Cilksan Output

```
$ cilksan ./nqueens 9
…
Race detected at address 0x7ffea4ab76b0
  Read access to  board_in (declared at nqueens/nqueens.c:54)
             from 0x4011cf nqueens nqueens/nqueens.c:65:3
       Called from 0x401a01 nqueens nqueens/nqueens.c:71:31
     Spawned from 0x4012de nqueens nqueens/nqueens.c:71:31
  Write access to board (declared at nqueens/nqueens.c:56)
             from 0x40128d nqueens nqueens/nqueens.c:68:16
  Common calling context
       Called from 0x401703 main nqueens/nqueens.c:104:9

0.269
Total

Race
Race
```

```
65    memcpy(board, board_in, row * sizeof(char));
66    // Try each column for placing the queen.
67    for (int col = 0; col < n; col++) {
68      board[row] = col;
69      // If this board is OK, continue search in parallel
70      if (board_ok(row + 1, board))
71        count[col] = cilk_spawn nqueens(n, row + 1, board);
72    }
73    cilk_sync;
```

Correct Queens Code

```
int nqueens(int n, int row, char *board_in) {
  if (n == row) return 1;  // Base case: Fully populated board
  char *board;
  int count[n];
  memset(count, 0, n * sizeof(int));

  // Try each column for placing the queen.
  for (int col = 0; col < n; col++) {
    // Make a local copy of the board
    board = (char *)alloca((row + 1) * sizeof(char));
    memcpy(board, board_in, row * sizeof(char));

    board[row] = col;
    // If this board is OK, continue search in parallel
    if (board_ok(row + 1, board))
      count[col] = cilk_spawn nqueens(n, row + 1, board);
  }
  cilk_sync;
  // Return the total number of solutions found.
  return total_solutions(count, n);
}
```

Avoiding Races

- Iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children.
  - **Note:** The arguments to a spawned function are evaluated in the parent before the spawn occurs.
- Machine word size matters. Watch out for races in packed data structures:

```
struct {
  char a;
  char b;
} x;
```

Ex. Updating `x.a` and `x.b` in parallel may cause a race! Nasty, because it may depend on the compiler optimization level. (Safe on Intel x86-64.)