Software
Performance
Engineering

LECTURE 2

Bit Hacks

Xuhao Chen

Thursday, August 28, 2025

C Bitwise Operators

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

C Bitwise Operators

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

Examples (8-bit word)

```
A = 0b10110011
B = 0b01101001

A & B = 0b00100001        ~A = 0b01001100
A | B = 0b11111011    A >> 3 = 0b00010110
A ^ B = 0b11011010    A << 2 = 0b11001100
```

C Bitwise Operators

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

The prefix 0b designates a Boolean constant.

Examples (8-bit word)

```
A = 0b10110011
B = 0b01101001

A & B = 0b00100001
A | B = 0b11111011
A ^ B = 0b11011010

~A = 0b01001100
A >> 3 = 0b00010110
A << 2 = 0b11001100
```

Integer Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w-bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

For example, the 8-bit word `0b01101010` represents the unsigned value `106 = 2 + 8 + 32 + 64`.

The signed integer (two's complement) value stored in `x` is

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1} .$$

For example, the 8-bit word `0b10010110` represents the signed value `-106 = 2 + 4 + 16 - 128`.

Elementary Notions

Simple properties

- Add, subtract, multiply, and divide binary numbers.
- Hexadecimal/binary conversion.
- A word with all `0` bits equals `0`.
- A word with all `1` bits equals `-1`.
- `~x + x = -1` and `-x = ~x + 1`.
- Etc.

Masking operations

- Set the `k`th bit.
- Clear the `k`th bit.
- Toggle the `k`th bit.
- Extract a bit field.
- Set a bit field.

Example

```
 8-bit word: abcdefgh
 8-bit mask: 10011101
word & mask: a00def0h
```

Powers of 2

Least-Significant 1

Problem

Compute the mask of the least-significant 1 in word x.
Equivalently, what is the largest power of 2 that divides x?

```
r = x & (-x);
```

Example

x	0010000001010000
-x	1101111110110000
x & (-x)	0000000000010000

Why it works

- `-x = ~x + 1`.

Question

How do you find the index of the bit, i.e., $\lg r$?

Notation: $\lg r = \log_2 r$

Is an Integer a Power of 2?

Problem

Is $x = 2^k$ for some integer k?

```
x == x & -x
```

Example

x	00001000	00101000
-x	11111000	11011000
x & -x	00001000	00001000
x == x & -x	00000001	00000000

Bug!

What if $x = 0$?

```
(x != 0) && (x == x & -x)
```

Count Trailing Zeros

Problem

Compute `lg x`, where `x` is a power of `2`.

```
const uint64_t deBruijn = 0x022fdd63cc95386d;
const int convert[64] = {
    0,  1,  2, 53,  3,  7, 54, 27,
    4, 38, 41,  8, 34, 55, 48, 28,
   62,  5, 39, 46, 44, 42, 22,  9,
   24, 35, 59, 56, 49, 18, 29, 11,
   63, 52,  6, 26, 37, 40, 33, 47,
   61, 45, 43, 21, 23, 58, 17, 10,
   51, 25, 36, 32, 60, 20, 57, 16,
   50, 31, 19, 15, 30, 14, 13, 12
};
r = convert[(x * deBruijn) >> 58];
```

Count Trailing 0's of a Power of 2

Why it works

A deBruijn sequence s of length 2^k is a cyclic 0-1 sequence such that each of the 2^k 0-1 strings of length k occurs exactly once as a substring of s.

Example: $k=3$

	00011101
0	00011101
1	00111010
2	01110100
3	11101000
4	11010001
5	10100011
6	01000111
7	10001110

Count Trailing 0's of a Power of 2

Why it works

A deBruijn sequence `s` of length 2^k is a cyclic `0-1` sequence such that each of the 2^k `0-1` strings of length `k` occurs exactly once as a substring of `s`.

Example: k=3

	00011101
0	00011101
1	00111010
2	01110100
3	11101000
4	11010001
5	10100011
6	01000111
7	10001110

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

Count Trailing 0's of a Power of 2

Why it works

A deBruijn sequence `s` of length 2^k is a cyclic `0-1` sequence such that each of the 2^k `0-1` strings of length `k` occurs exactly once as a substring of `s`.

`0b00011101*2`^{4} ⇒ `0b11010000`

`0b11010000 >> 5` ⇒ `110` ⇒ `6`

`convert[6]` ⇒ `4`

Example: k=3

	00011101
0	00011101
1	00111010
2	01110100
3	11101000
4	11010000
5	10100000
6	01000000
7	10000000

Hardware instruction

```
int __builtin_ctz(int x)
```

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

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111
0011111111111111
0100000000000000

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111
0011111111111111
0100000000000000

Why decrement?

To handle the boundary case when n is a power of 2.

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Bit $\lceil \lg n \rceil - 1$ must be set

Set bit $\lceil \lg n \rceil$

Populate all the bits to the right with 1

Example

0010000001010000
0010000001001111
0011000001101111
0011111001111111
0011111111111111
0100000000000000

SWAPPING

Ordinary Swap

Problem

Swap two integers x and y.

Ordinary Swap

Problem

Swap two integers x and y.

```
t = x;
x = y;
y = t;
```

Example

x	10111101	10111101	00101110	10111101
y	00101110	00101110	00101110	00101110
t		10111101	10111101	10111101

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101			
y	00101110			

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011		
y	00101110	00101110		

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	
y	00101110	00101110	10111101	

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

(x ^ y) ^ y

(x ^ y) ^ x

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse:

`(x ^ y) ^ y ⇒ x`

x	y	x ^ y	(x ^ y) ^ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101			
y	00101110			

Why it works

XOR is its own inverse:

`(x ^ y) ^ y ⇒ x`

x	y	x ^ y	(x ^ y) ^ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

Mask with 1's where bits differ

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011		
y	00101110	00101110		

Why it works

XOR is its own inverse:

`(x ^ y) ^ y ⇒ x`

x	y	x ^ y	(x ^ y) ^ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

Flip bits in y that differ from x

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	
y	00101110	00101110	10111101	

Why it works

XOR is its own inverse:

$(x \oplus y) \oplus y \Rightarrow x$

x	y	x ^ y	(x ^ y) ^ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

Flip bits in x that differ from y

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse:

$(x \oplus y) \oplus y \Rightarrow x$

x	y	x ^ y	(x ^ y) ^ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse: `(x ^ y) ^ y` ⇒ `x`

Performance 👎

Poor at exploiting instruction-level parallelism (ILP).

Avoiding Unpredictable Code Branches

Minimum of Two Integers

Problem

Find the minimum `r` of two integers `x` and `y`.

```
if (x < y)
  r = x;
else
  r = y;
```

or

```
r = (x < y) ? x : y;
```

Performance

A mispredicted branch empties the processor pipeline.

Caveat

The compiler is usually smart enough to optimize away the unpredictable branch, but maybe not.

No-Branch Minimum

Problem

Find the minimum of two integers x and y without using a branch:

```
y ^ ((x ^ y) & -(x < y));
```

Why it works

- The C language represents the Booleans TRUE and FALSE with the integers 1 and 0, respectively.
- If $x < y$, then $-(x < y) = -1$, which is all 1's in two's complement representation. Therefore, we have $y \wedge ((x \wedge y) \,\&\, 1) = y \wedge (x \wedge y) = x$.
- If $x \geq y$, then $y \wedge ((x \wedge y) \,\&\, 0) = y \wedge 0 = y$.

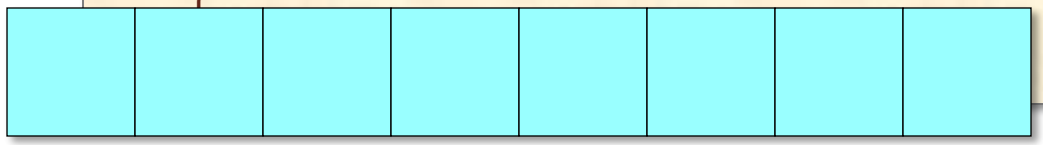
Merging Two Sorted Arrays

```
static void merge(int64_t * __restrict C,
                  int64_t * __restrict A,
                  int64_t * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
}
```

The `__restrict` keywords say that **A**, **B**, and **C** don't alias, meaning that they do not overlap in memory.

Merging Two Sorted Arrays

```
static void merge(int64_t * __restrict C,
                  int64_t * __restrict A,
                  int64_t * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
```



3	12	19	46

4	14	21	23

Branching

```
static void merge(long * __restrict C,
                  long * __restrict A,
                  long * __restrict B,
                  size_t na,
                  size_t nb) {
4   while (na > 0 && nb > 0) {
3     if (*A <= *B) {
        *C++ = *A++; na--;
      } else {
        *C++ = *B++; nb--;
      }
    }
2   while (na > 0) {
      *C++ = *A++;
      na--;
    }
1   while (nb > 0)  {
      *C++ = *B++;
      nb--;
    }
}
```

Branch	Predictable?
1	Yes
2	Yes
3	No
4	Yes

Branchless

```
static void merge(int64_t * __restrict C,
                  int64_t * __restrict A,
                  int64_t * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    long cmp = (*A <= *B);
    long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A +=  cmp; na -=  cmp;
    B += !cmp; nb -= !cmp;
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
}
```

On modern machines using `clang -O3`, the branchless version is usually slower than the branching version 👎 because modern compilers convert unpredictable branches into predictable ones using **conditional-move** hardware instructions.

Modular Addition

Problem

Compute $r = (x + y) \bmod n$, assuming that $0 \le x < n$ and $0 \le y < n$.

```
r = (x + y) % n;
```

Division is expensive.

```
z = x + y;
r = (z < n) ? z : z - n;
```

An unpredictable branch is expensive.

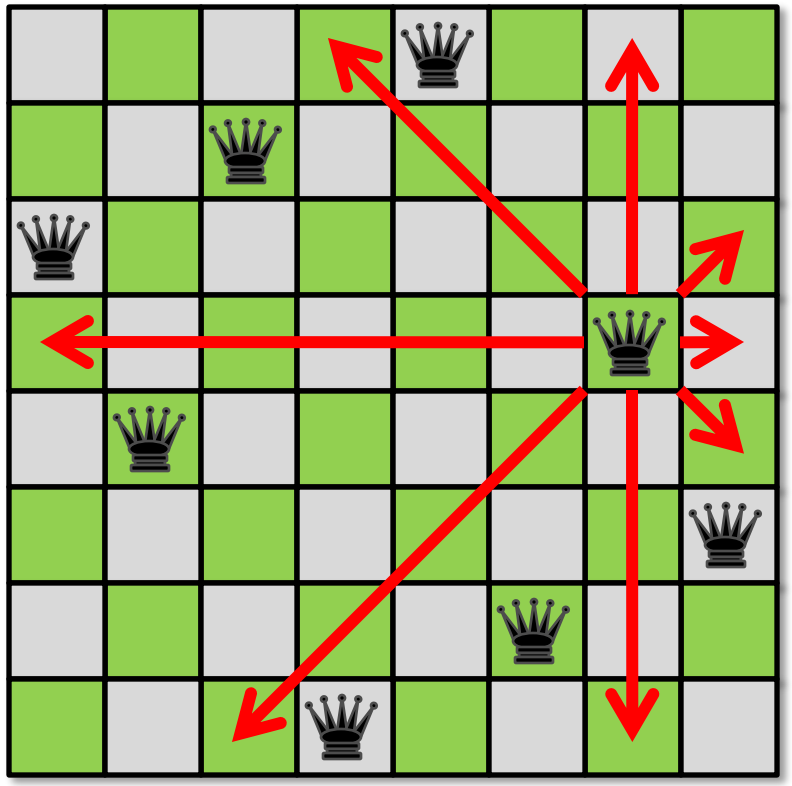
```
z = x + y;
r = z - (n & -(z >= n));
```

Same trick as minimum.

The Queens Problem

Queens Problem

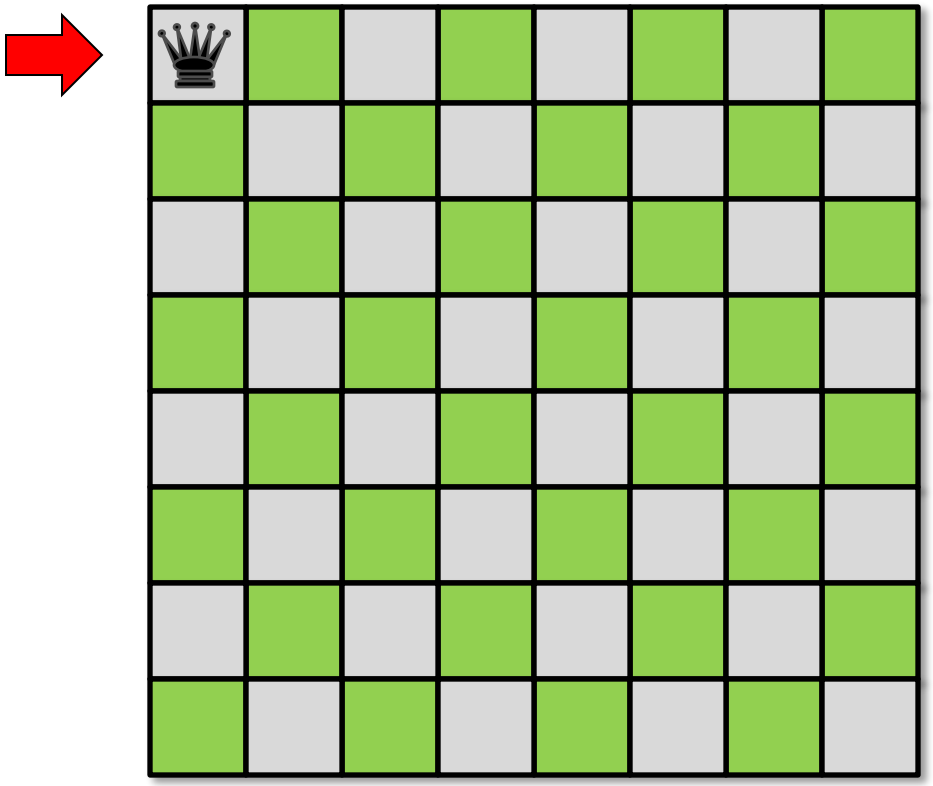
Place n Queens on an $n \times n$ chessboard so that no Queen attacks another, i.e., no two Queens in any row, column, or diagonal. Count the number of possible solutions.



Backtracking Search

Strategy

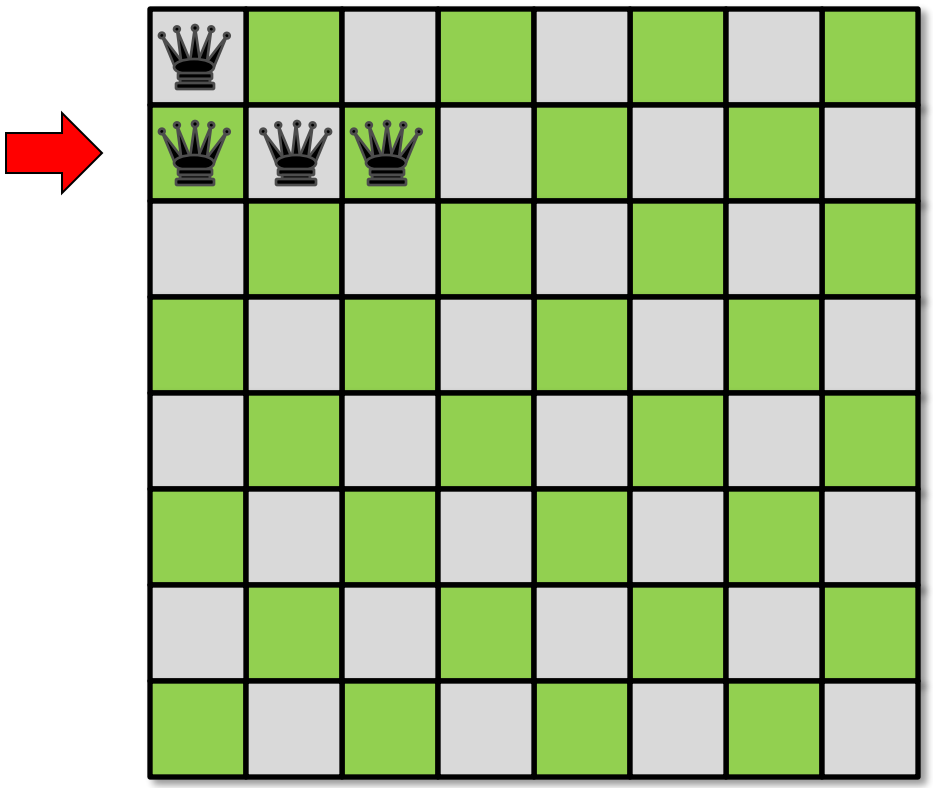
Try placing Queens row by row. If you can't place a Queen in a row, backtrack.



Backtracking Search

Strategy

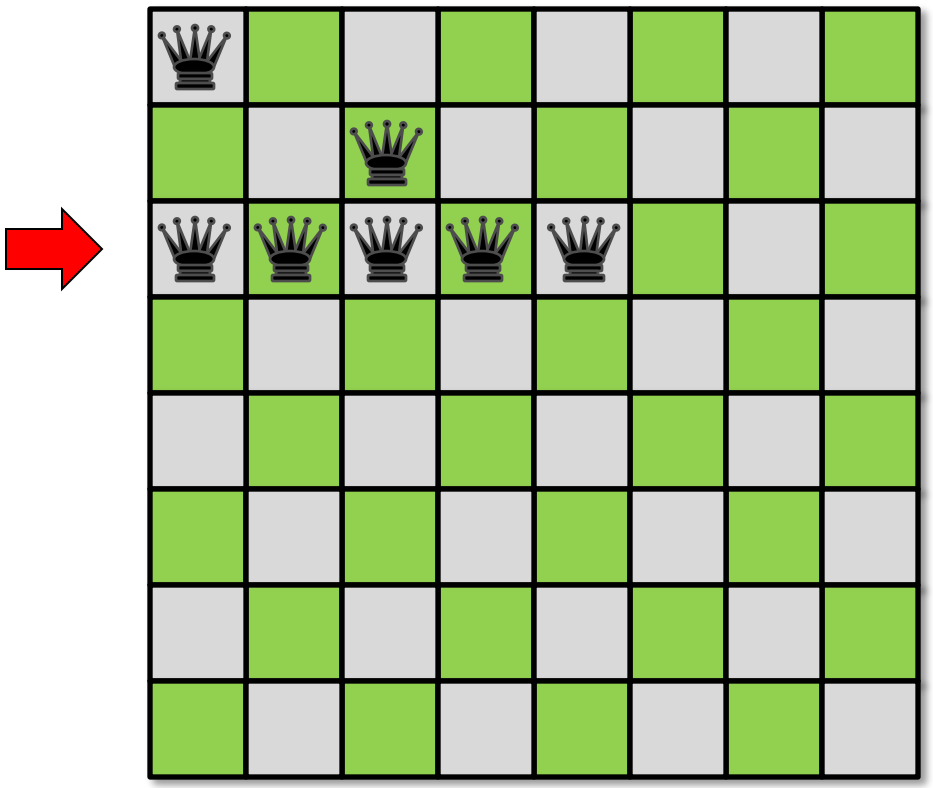
Try placing Queens row by row. If you can't place a Queen in a row, backtrack.



Backtracking Search

Strategy

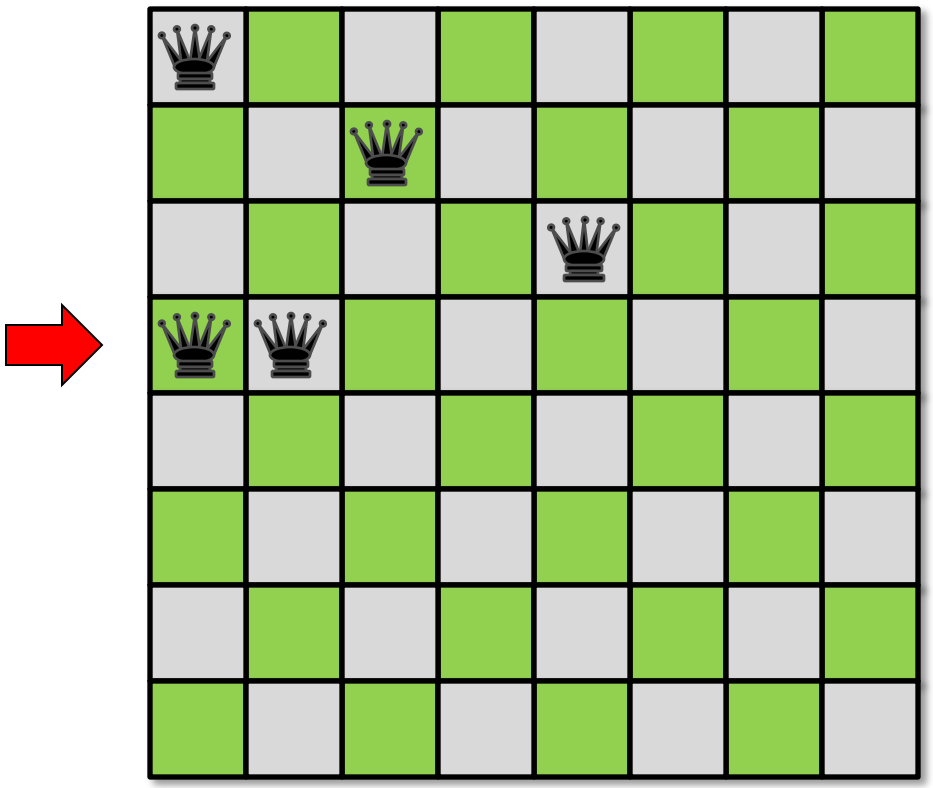
Try placing Queens row by row. If you can't place a Queen in a row, backtrack.



Backtracking Search

Strategy

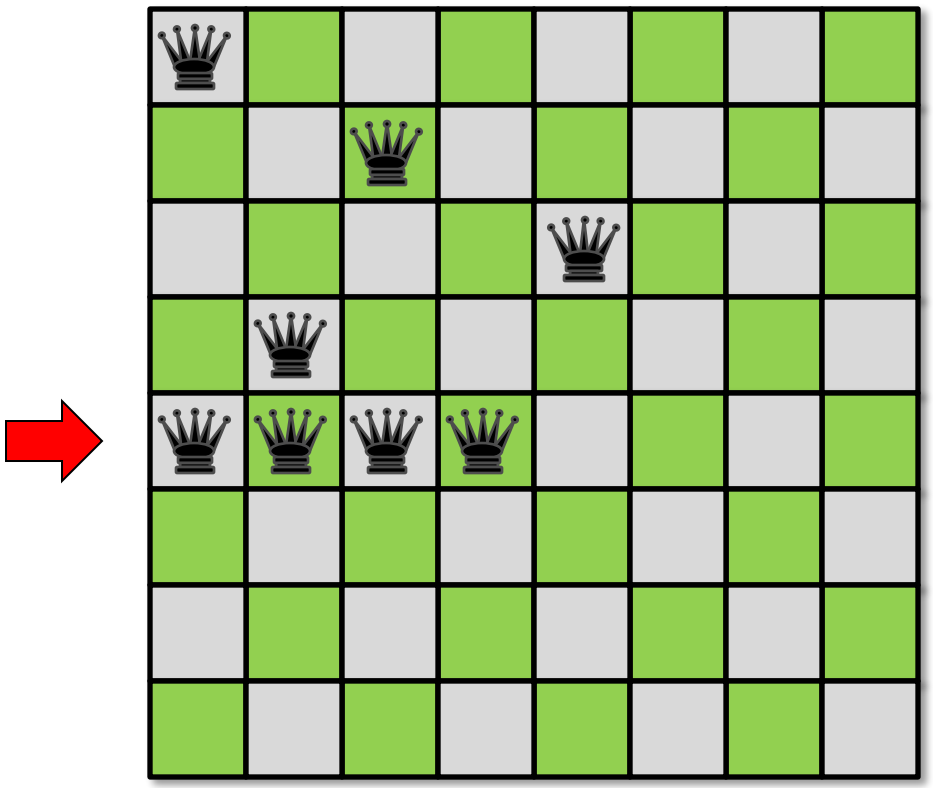
Try placing Queens row by row. If you can't place a Queen in a row, backtrack.



Backtracking Search

Strategy

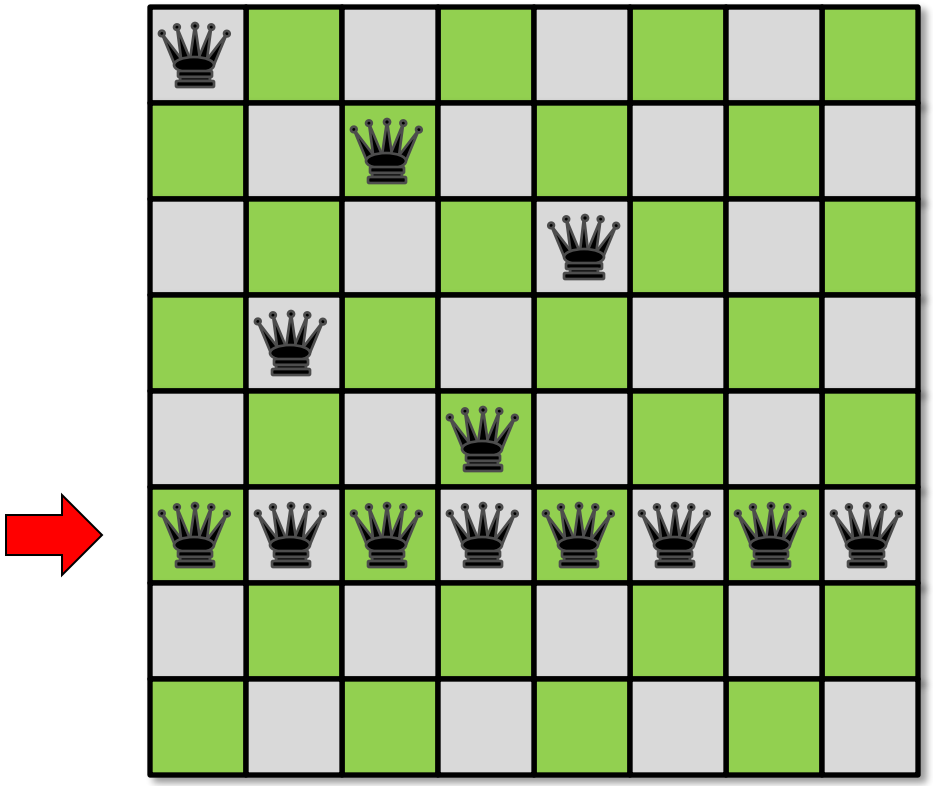
Try placing Queens row by row. If you can't place a Queen in a row, backtrack.



Backtracking Search

Strategy

Try placing Queens row by row. If you can't place a Queen in a row, backtrack.

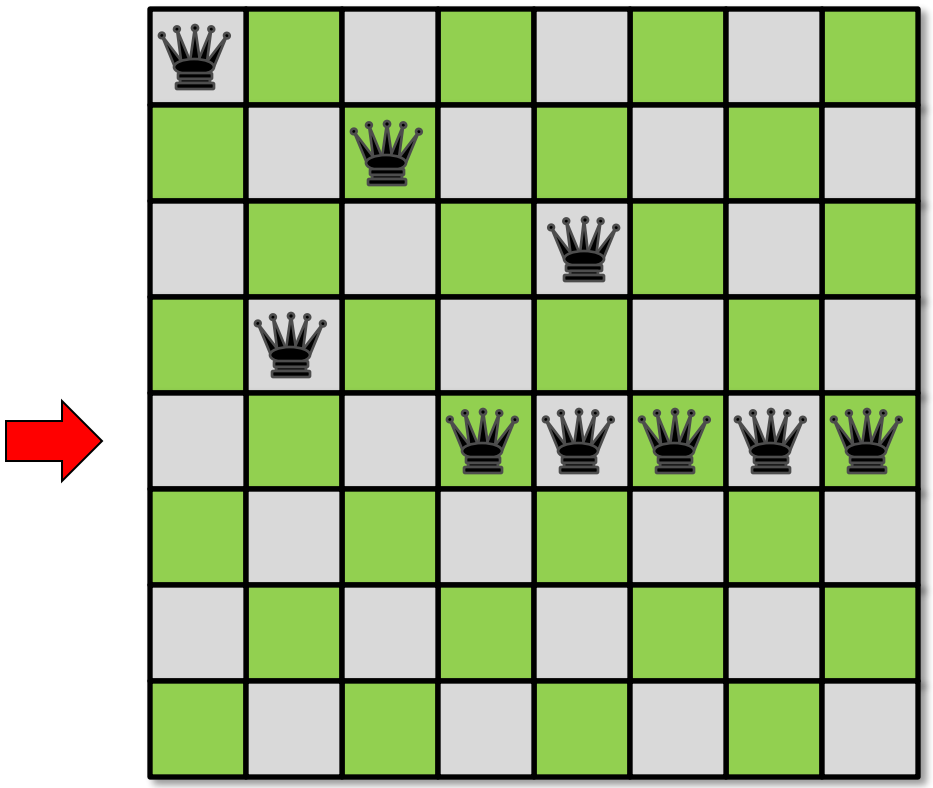


Backtrack!

Backtracking Search

Strategy

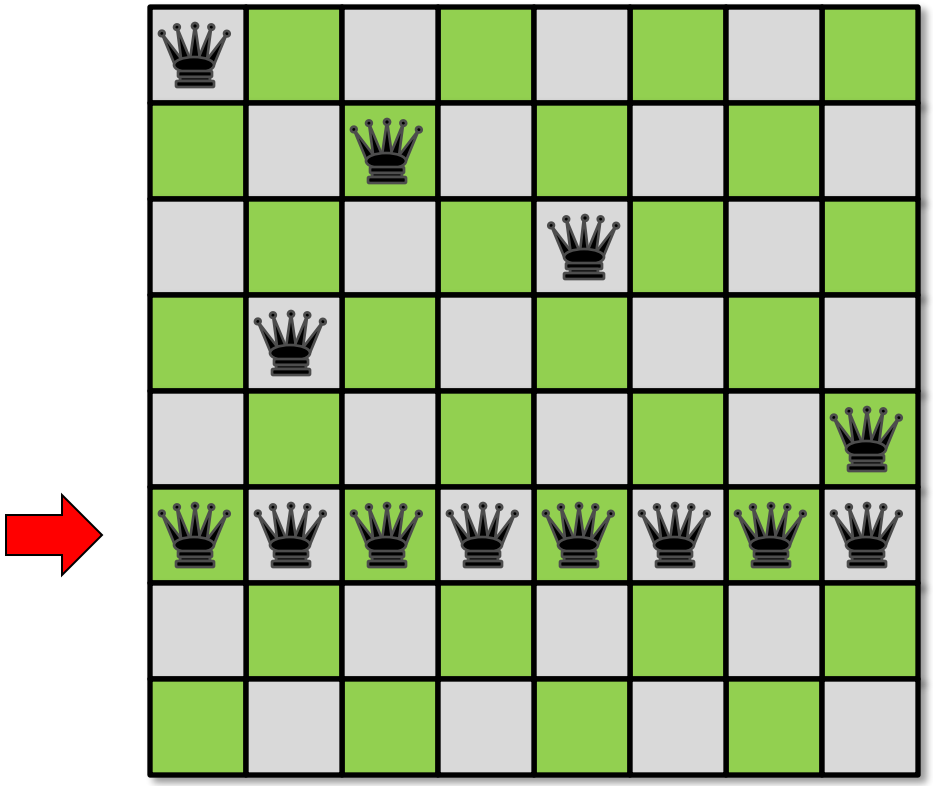
Try placing Queens row by row. If you can't place a Queen in a row, backtrack.



Backtracking Search

Strategy

Try placing Queens row by row. If you can't place a Queen in a row, backtrack.

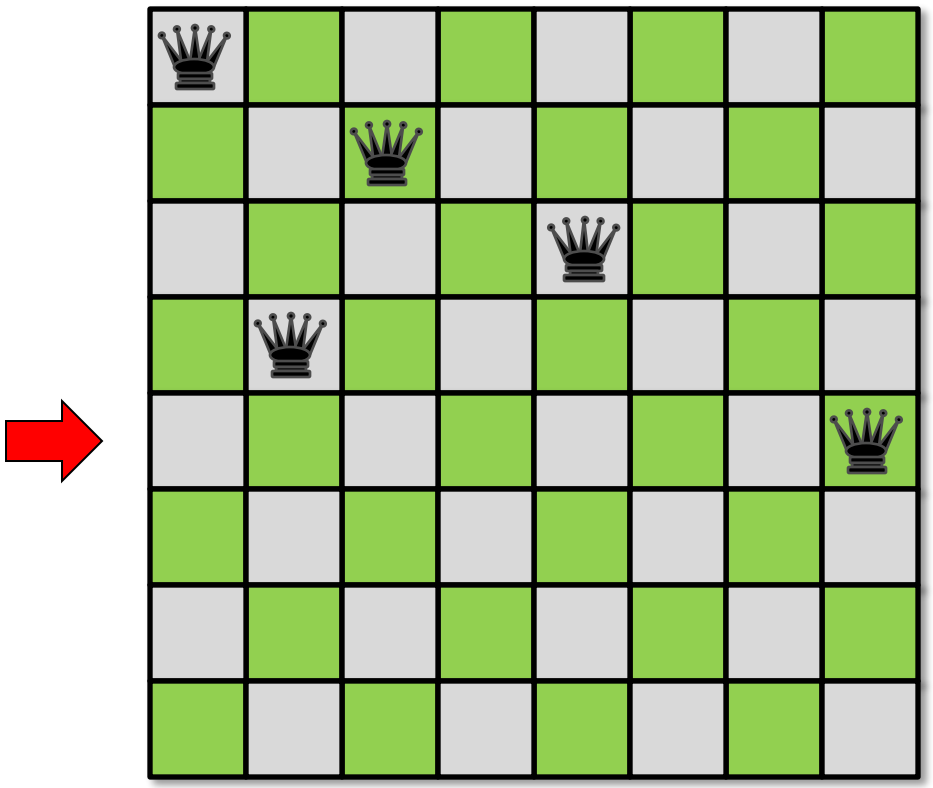


Backtrack!

Backtracking Search

Strategy

Try placing Queens row by row. If you can't place a Queen in a row, backtrack.

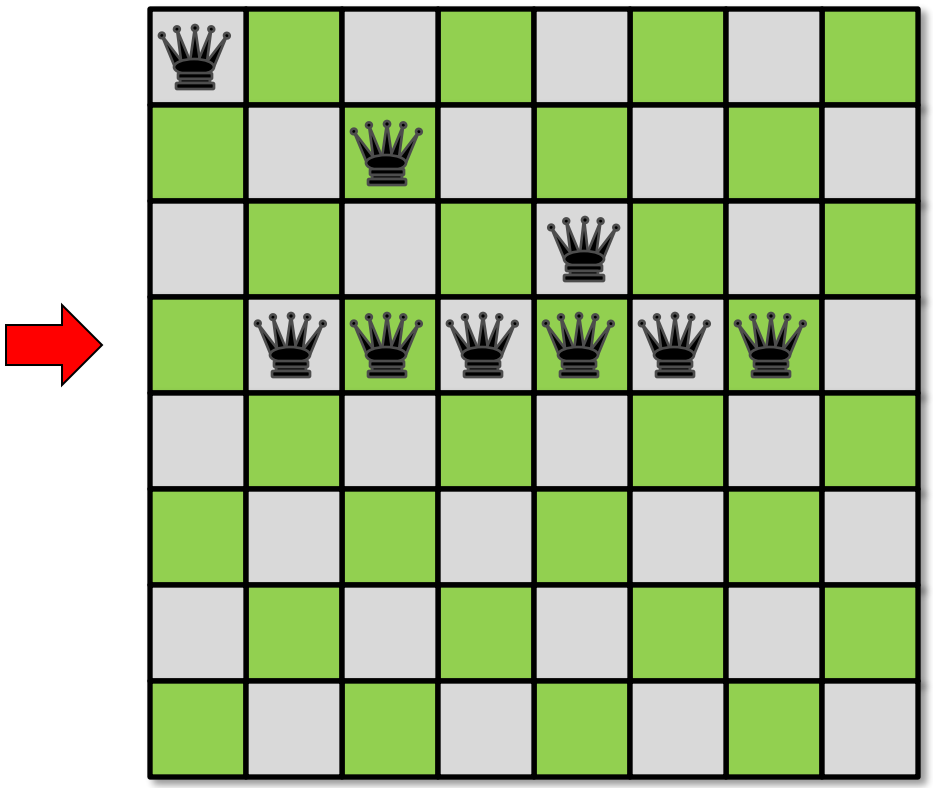


Backtrack!

Backtracking Search

Strategy

Try placing Queens row by row. If you can't place a Queen in a row, backtrack.



Board Representation

The backtrack search can be implemented as a simple recursive procedure, but how should the board be represented to facilitate Queen placement?

- array of n^2 bytes?
- array of n^2 bits?
- array of n bytes?
- a few bitvectors of size n.

Queens Code

Code inspired by Tony Lezard (1991)

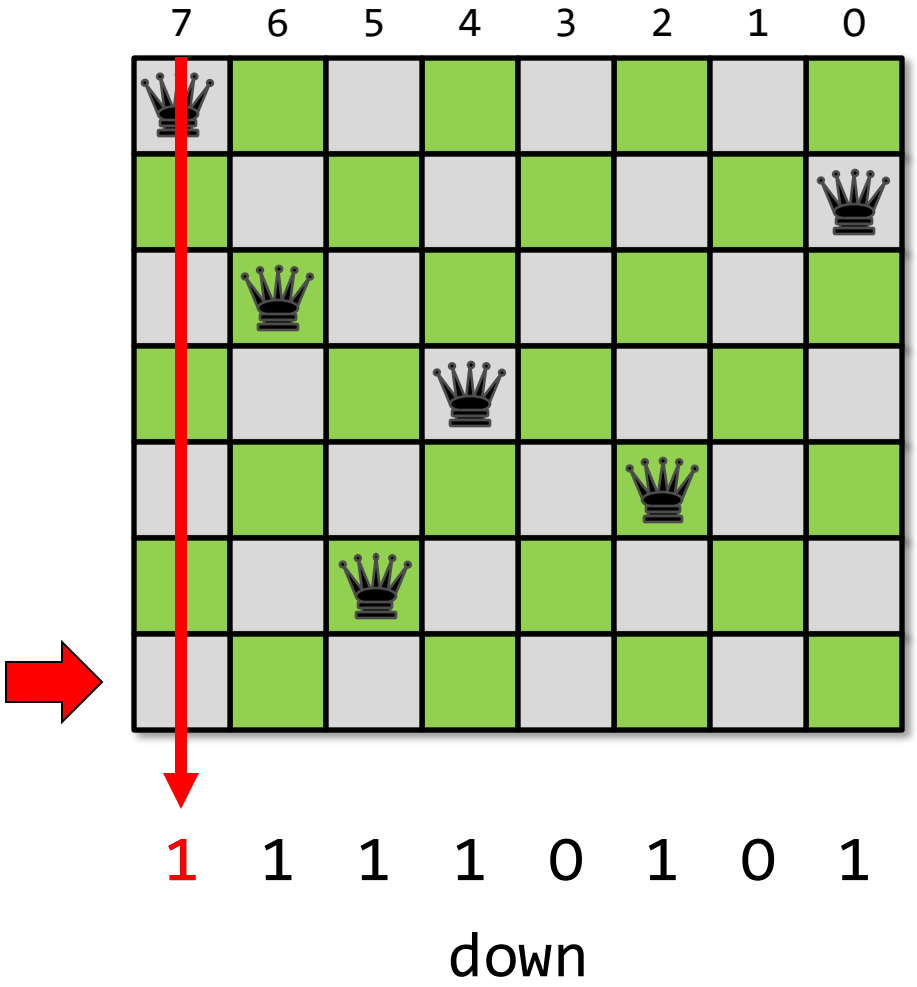
```
int32_t
queens(uint32_t mask,
       uint32_t down,
       uint32_t left,
       uint32_t right) {
  int32_t possible, place;
  int32_t count = 0;
  if (down == mask) return 1;
  for (possible = ~(down | left | right) & mask;
       possible != 0;
       possible &= ~place) {
    place = possible & -possible;
    count += queens(mask,
                    down | place,
                    ((left | place) << 1) & mask,
                    (right | place) >> 1);
  }
  return count;
}
```

Root call

```
queens((1 << n)-1, 0, 0, 0));
```

Tony Lezard

Bitvector Representation

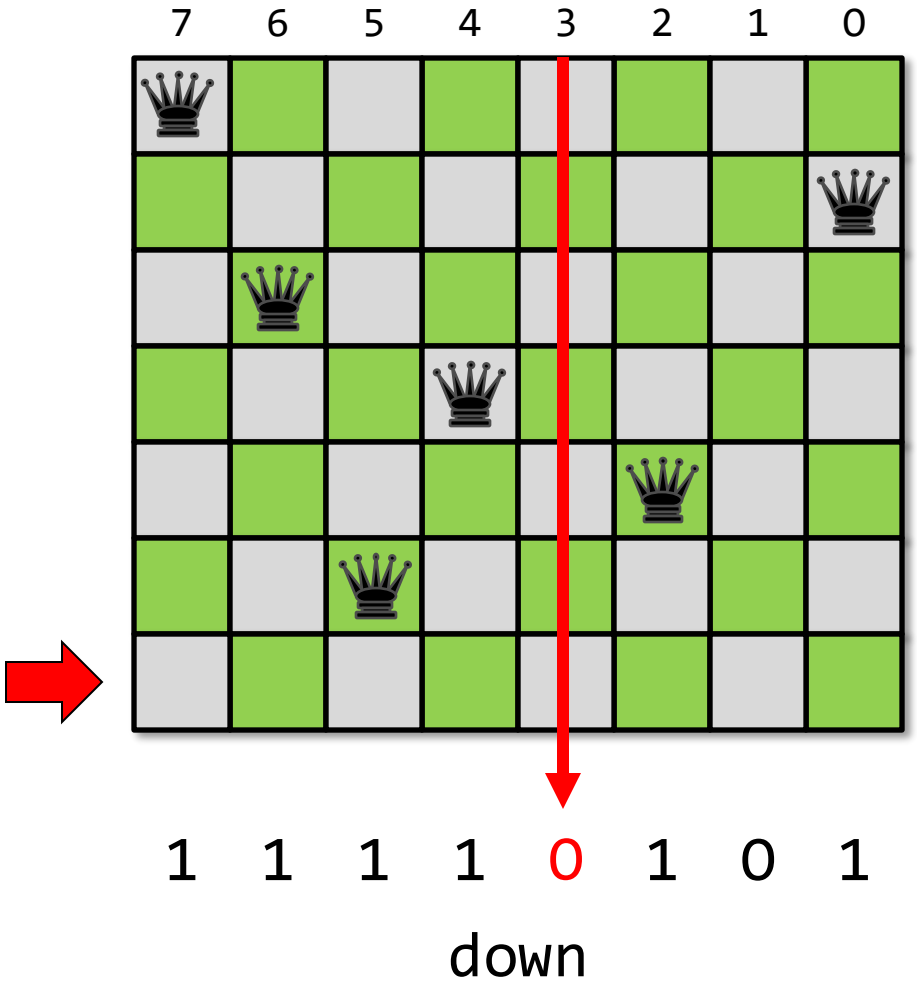


Placing a Queen in column `c` is not safe if

```
down & place != 0
```

where `place = 1 << c`.

Bitvector Representation

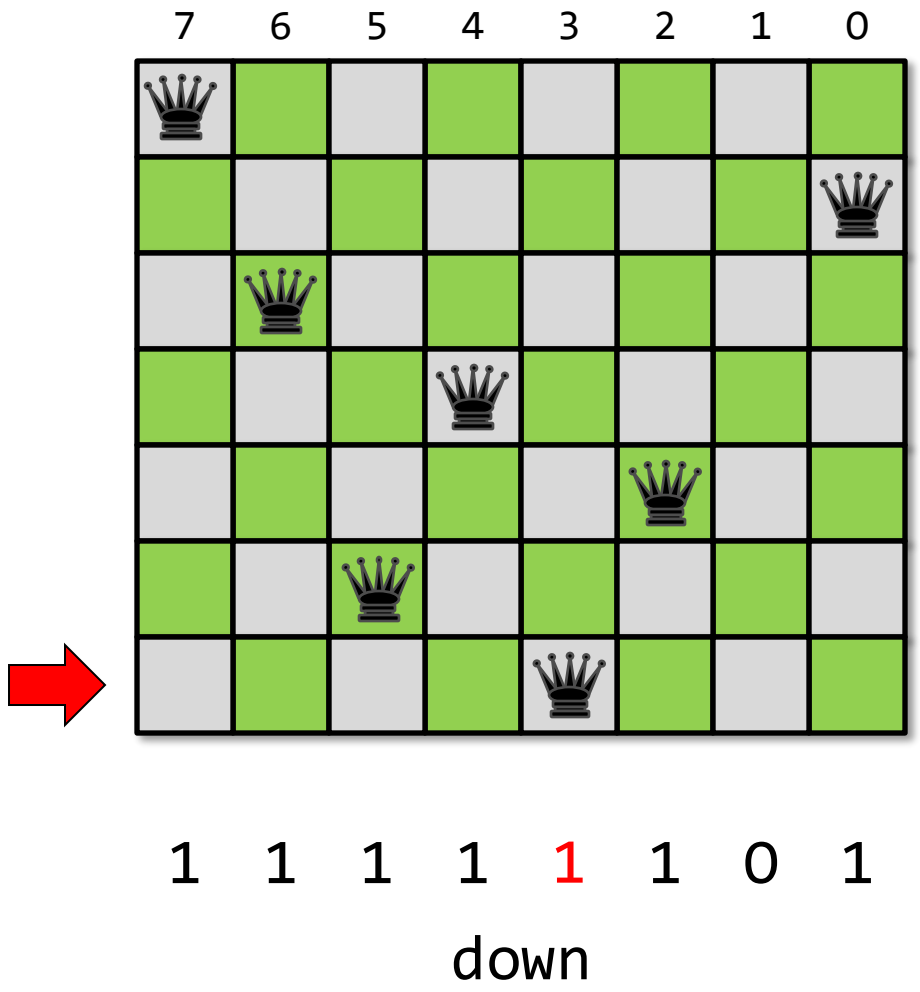


Placing a Queen in column `c` is not safe if

```
down & place != 0
```

where `place = 1 << c`.

Bitvector Representation



Placing a Queen in column `c` is not safe if

`down & place != 0`

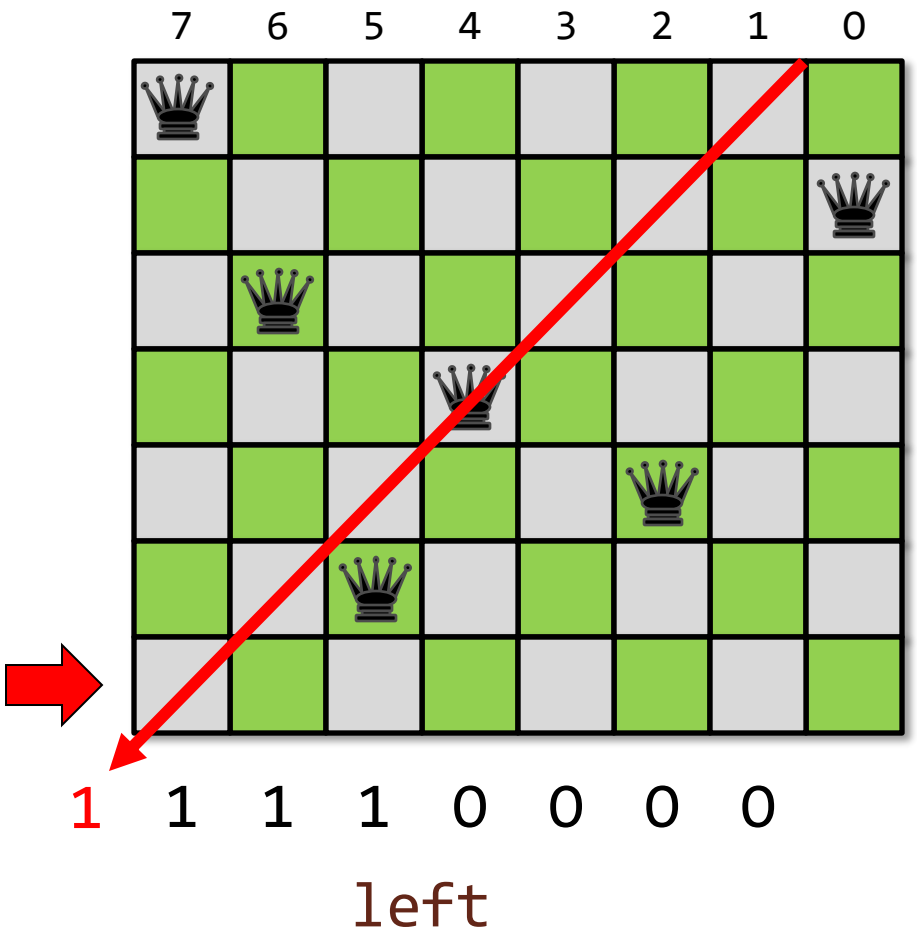
where `place = 1 << c`.

If column `c` is safe, then update

`down |= place`

for the next row.

Bitvector Representation

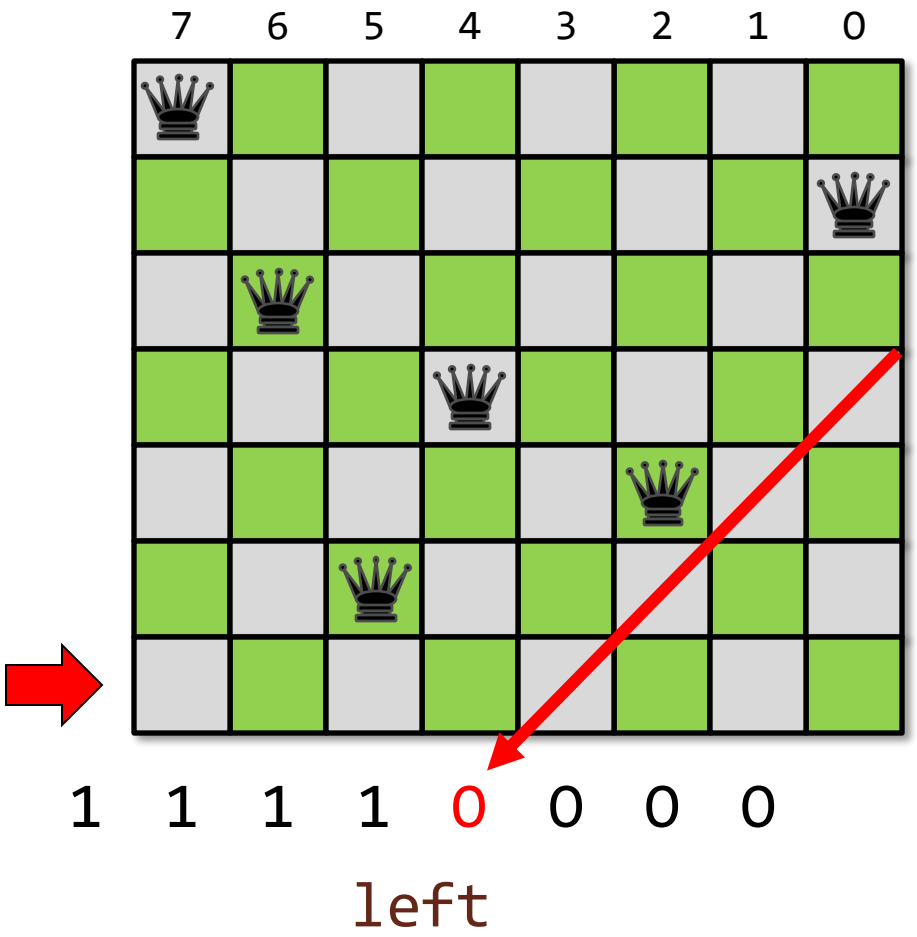


Placing a Queen in column `c` is not safe if

```
left & place != 0
```

where `place = 1 << c`.

Bitvector Representation

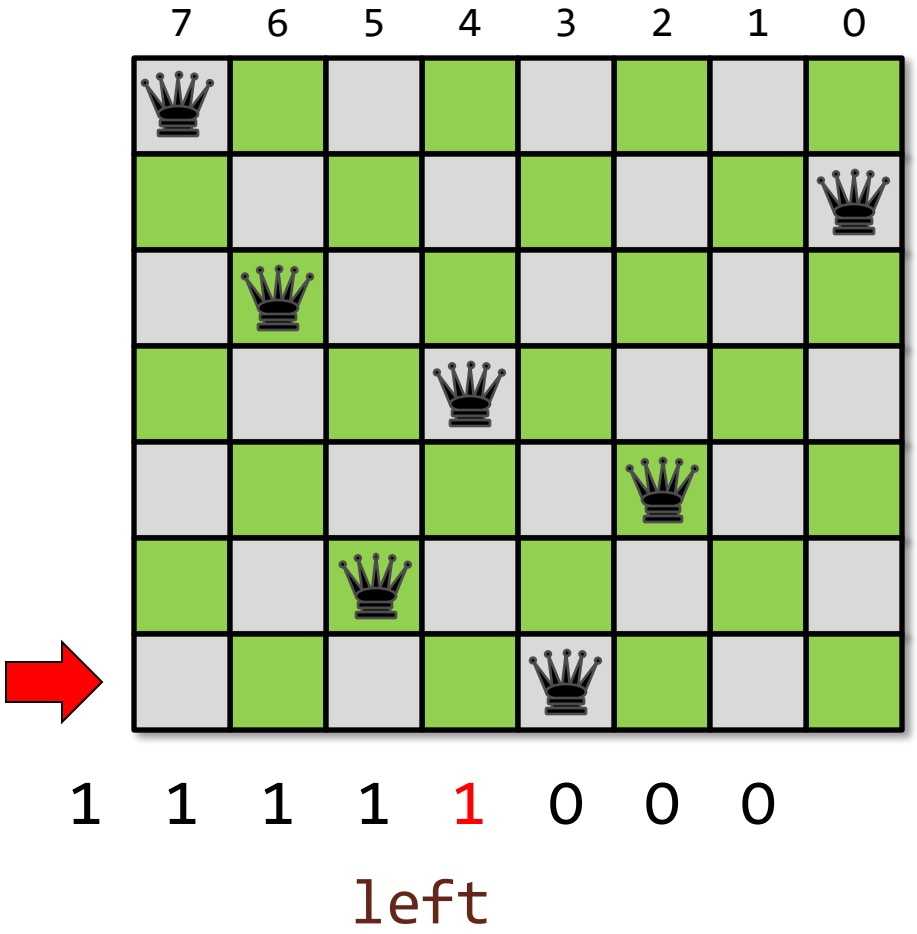


Placing a Queen in column `c` is not safe if

```
left & place != 0
```

where `place = 1 << c`.

Bitvector Representation



Placing a Queen in column `c` is not safe if

```
left & place != 0
```

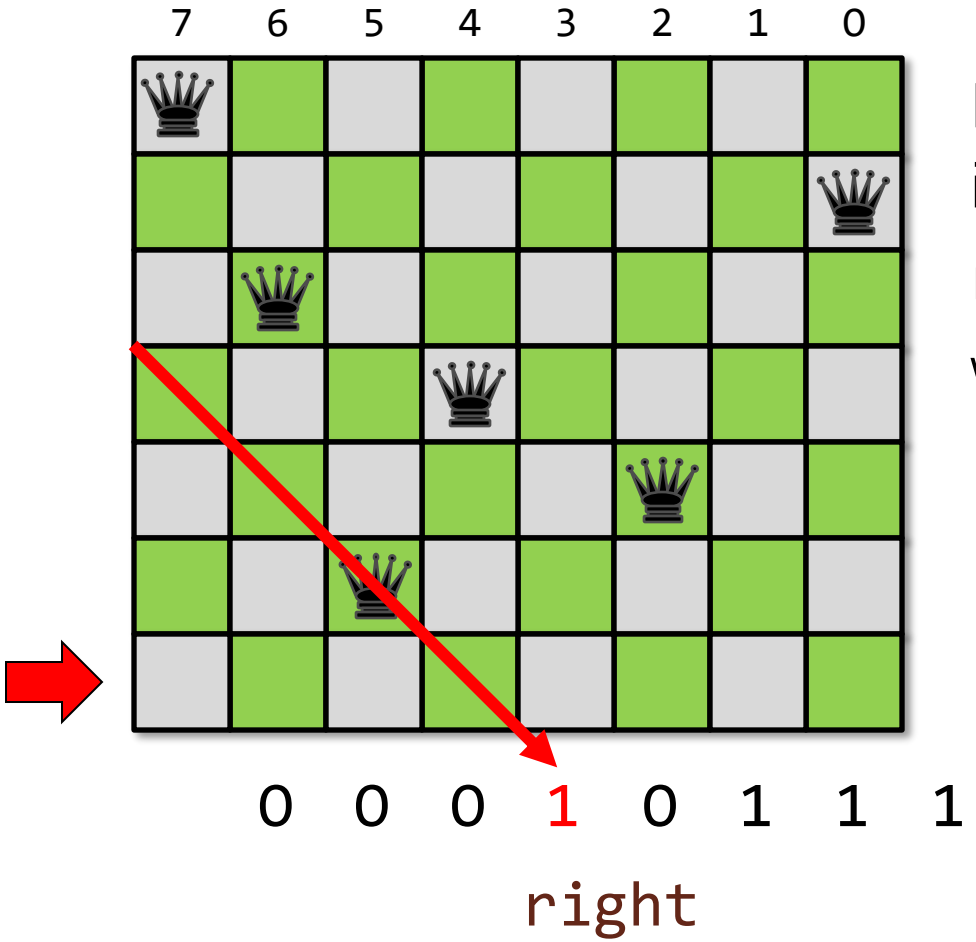
where `place = 1 << c`.

If column `c` is safe, then update

```
left = (left|place) << 1
```

for the next row.

Bitvector Representation

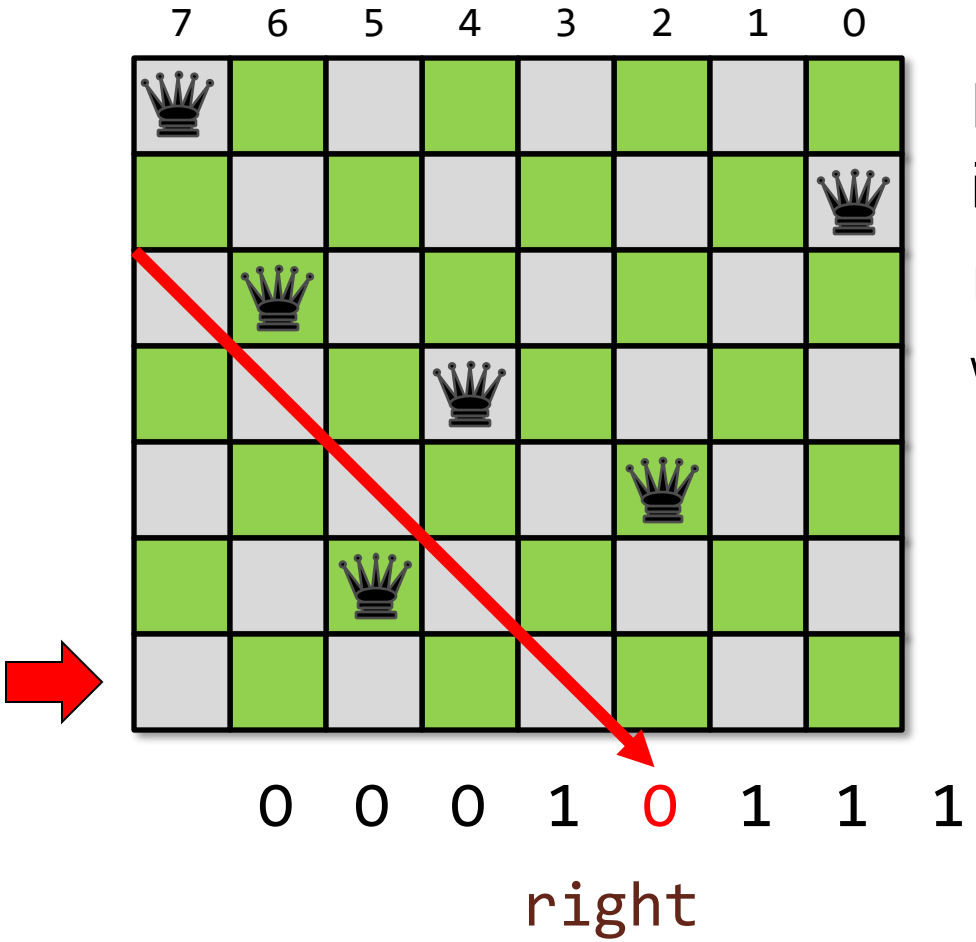


Placing a Queen in column `c` is not safe if

`right & place != 0`

where `place = 1 << c`.

Bitvector Representation

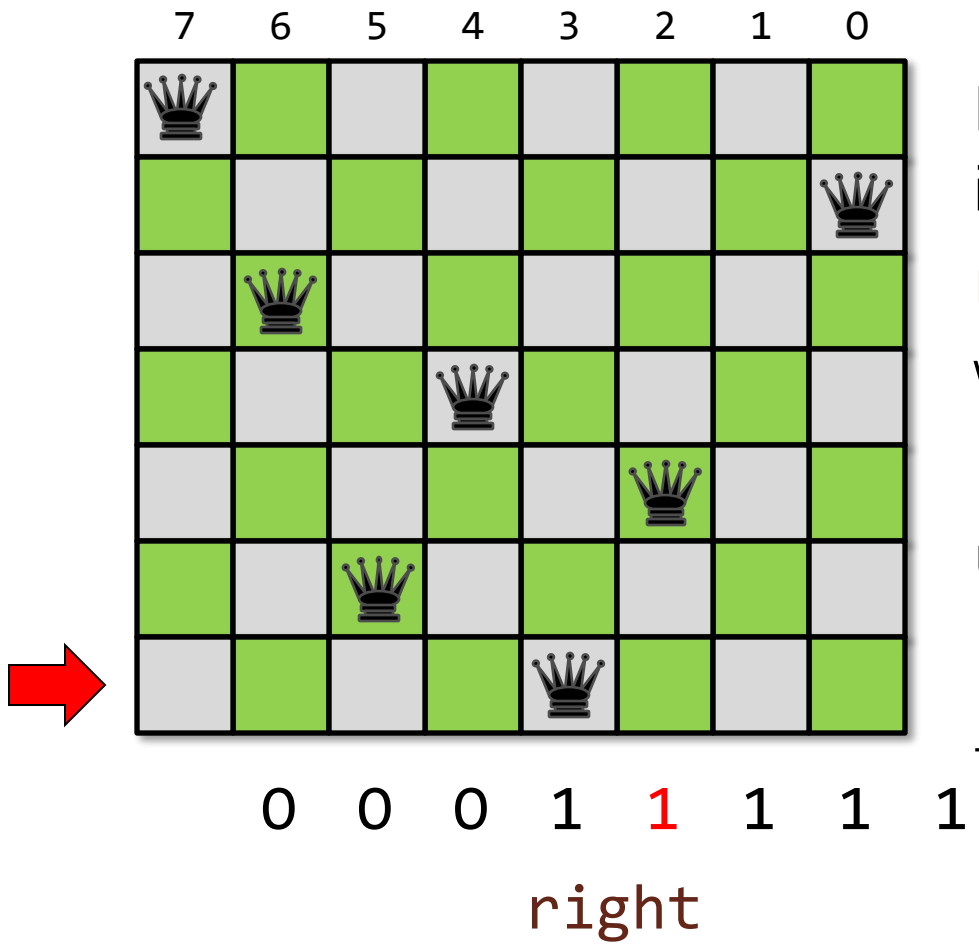


Placing a Queen in column `c` is not safe if

`right & place != 0`

where `place = 1 << c`.

Bitvector Representation



Placing a Queen in column `c` is not safe if

`right & place != 0`

where `place = 1<<c`.

If column `c` is safe, then update

`right = (right|place)<<1`

for the next row.

Queens Code

Code inspired by Tony Lezard (1991)

```
int32_t
queens(uint32_t mask,
       uint32_t down,
       uint32_t left,
       uint32_t right) {
  int32_t possible, place;
  int32_t count = 0;
  if (down == mask) return 1;
  for (possible = ~(down | left | right) & mask;
       possible != 0;
       possible &= ~place) {
    place = possible & -possible;
    count += queens(mask,
                    down | place,
                    ((left | place) << 1) & mask,
                    (right | place) >> 1);
  }
  return count;
}
```

Root call

```
queens((1 << n)-1, 0, 0, 0));
```

Tony Lezard

Final Remarks

Further Reading

- Sean Eron Anderson, "Bit twiddling hacks," http://graphics.stanford.edu/~seander/bithacks.html, 2009.
- Donald E. Knuth, The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1, Addison-Wesley, 2011, Section 7.1.3.
- http://chessprogramming.wikispaces.com/
- Henry S. Warren, Hacker's Delight, Addison-Wesley, 2003.

And remember…

Every Little Bit Counts!

Appendix A: Binary Representations

Unsigned-Integer Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w-bit computer word.
The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

Unsigned-Integer Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w-bit computer word.
The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

For example, the 8-bit word `0b01101010` represents the unsigned value $106 = 2 + 8 + 32 + 64$.

Unsigned-Integer Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w-bit computer word.
The unsigned integer value stored in

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

The prefix 0b designates a Boolean constant.

For example, the 8-bit word 0b01101010 represents the unsigned value $106 = 2 + 8 + 32 + 64$.

Integer Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w-bit computer word.
The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

For example, the 8-bit word `0b01101010` represents the unsigned value $106 = 2 + 8 + 32 + 64$.

The signed integer (two's complement) value stored in x is

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1} .$$

For example, the 8-bit word `0b10010110` represents the signed value $-106 = 2 + 4 + 16 - 128$.

Integer Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w-bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

For example, the 8-bit word `0b01101010` represents the unsigned value $106 = 2 + 8 + 32 + 64$.

The signed integer (two's complement) value stored in x is

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1} .$$

sign bit

For example, the 8-bit word `0b10010110` represents the signed value $-106 = 2 + 4 + 16 - 128$.

Two's Complement

We have $0b00\ldots0 = 0$.

What is the value of $x = 0b11\ldots1$?

$$
\begin{aligned}
x &= \left(\sum_{k=0}^{w-2} x_k 2^k\right) - x_{w-1} 2^{w-1} \\
&= \left(\sum_{k=0}^{w-2} 2^k\right) - 2^{w-1} \\
&= (2^{w-1} - 1) - 2^{w-1} \\
&= -1 \,.
\end{aligned}
$$

Complementary Relationship

Important identity

Since we have $\sim x + x = -1$, it follows that

$$\sim x + 1 = -x \ .$$

Complementary Relationship

Important identity

Since we have $\sim x + x = -1$, it follows that

$$\sim x + 1 = -x \ .$$

Example

$$\begin{aligned} x &= 0b0001100000011100 \\ \sim x &= 0b1110011111100011 \\ -x &= 0b1110011111100100 \end{aligned}$$

Complementary Relationship

Important identity

Since we have $\sim x + x = -1$, it follows that

$$\sim x + 1 = -x \ .$$

Example

$$\begin{aligned} x &= 0b0001100000011100 \\ \sim x &= 0b1110011111100011 \\ -x &= 0b1110011111100100 \end{aligned}$$

Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

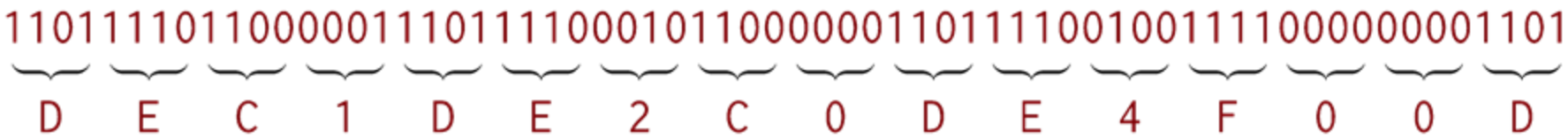
To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

Example: `0xDEC1DE2C0DE4F00D` is



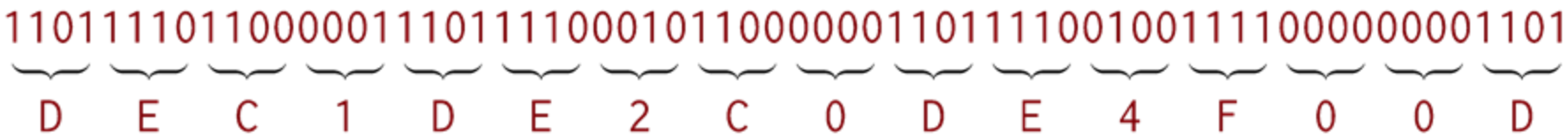
Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
	101	5	13	1101	D
	110	6	14	1110	E
	111	7	15	1111	F

The prefix 0x designates a hex constant.

To transla rom hex to binary, translate each hex digit to its nary equivalent, and concatenate the bits.

Example: 0xDEC1DE2C0DE4F00D is

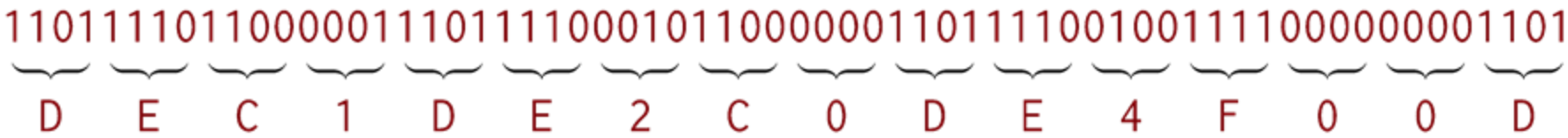


Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

Example: `0xDEC1DE2C0DE4F00D` is



Appendix B: Elementary Bit Hacks

C Bitwise Operators

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

Examples (8-bit word)

```
A = 0b10110011
B = 0b01101001

A & B = 0b00100001        ~A = 0b01001100
A | B = 0b11111011    A >> 3 = 0b00010110
A ^ B = 0b11011010    A << 2 = 0b11001100
```

Set the kth Bit

Problem

Set kth bit in a word x to 1.

Idea

Shift and OR.

```
x | (1 << k);
```

Set the kth Bit

Problem

Set kth bit in a word x to 1.

truth table for OR

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Idea

Shift and OR.

```
x | (1 << k);
```

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

Set the kth Bit

Problem

Set kth bit in a word x to 1.

truth table for OR

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Idea

Shift and OR.

```
x | (1 << k);
```

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

Set the kth Bit

Problem

Set kth bit in a word x to 1.

truth table for OR

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Idea

Shift and OR.

```
x | (1 << k);
```

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

Clear the kth Bit

Problem

Clear the kth bit in a word x.

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

Clear the kth Bit

Problem

Clear the k th bit in a word x.

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

truth table for AND

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

Clear the kth Bit

Problem

Clear the k th bit in a word x.

truth table for AND

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

Clear the kth Bit

Problem

Clear the k th bit in a word x.

truth table for AND

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

Toggle the kth Bit

Problem

Flip the k th bit in a word x.

Idea

Shift and XOR.

```
x ^ (1 << k);
```

Toggle the kth Bit

Problem

Flip the k th bit in a word x.

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Idea

Shift and XOR.

```
x ^ (1 << k);
```

Example (0 → 1)

k = 7

x	1011110101101101
1 << k	0000000010000000
x ^ (1 << k)	1011110111101101

Toggle the kth Bit

Problem

Flip the k th bit in a word x.

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Idea

Shift and XOR.

```
x ^ (1 << k);
```

Example (0 → 1)

k = 7

x	1011110101101101
1 << k	0000000010000000
x ^ (1 << k)	1011110111101101

Toggle the kth Bit

Problem

Flip the k th bit in a word x.

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Idea

Shift and XOR.

```
x ^ (1 << k);
```

Example (0 → 1)

k = 7

x	1011110101101101
1 << k	0000000010000000
x ^ (1 << k)	1011110111101101

Toggle the kth Bit

Problem

Flip the kth bit in a word x.

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Idea

Shift and XOR.

```
x ^ (1 << k);
```

Example (1 → 0)

k = 7

x	1011110111101101
1 << k	0000000010000000
x ^ (1 << k)	1011110101101101

Extract a Bit Field

Problem

Extract a bit field from a word `x`.

Idea

Mask and shift.

```
(x & mask) >> shift;
```

Extract a Bit Field

Problem

Extract a bit field from a word `x`.

Idea

Mask and shift.

```
(x & mask) >> shift;
```

Example

`shift = 7`

`x`	1011110101101101
`mask`	0000011110000000
`x & mask`	0000010100000000
`(x & mask) >> shift`	0000000000001010

Set a Bit Field

Problem

Set a bit field in a word `x` to a value `y`.

Idea

Invert mask to clear, and OR the shifted value.

```
(x & ~mask) | (y << shift);
```

Set a Bit Field

Problem

Set a bit field in a word `x` to a value `y`.

Idea

Invert mask to clear, and OR the shifted value.

```
(x & ~mask) | (y << shift);
```

Example

`shift = 7`

`x`	1011110101101101
`y`	0000000000000011
`mask`	0000011110000000
`x & ~mask`	1011100001101101
`y << shift`	0000000110000000
`(x & ~mask) | (y << shift)`	1011100111101101

Set a Bit Field Dangerously

Problem

Set a bit field in a word `x` to a value `y`.

Idea

Invert mask to clear, and OR the shifted value.

```
(x & ~mask) | (y << shift);
```

Dangerous example

`shift = 7`

`x`	1000110101101101
`y`	0000000000100011
`mask`	0000011110000000
`x & ~mask`	1000100001101101
`y << shift`	0001000110000000
`(x & ~mask) | (y << shift)`	1001100111101101

Set a Bit Field Safely

Problem

Set a bit field in a word `x` to a value `y` safely.

Idea

Invert mask to clear, and OR the masked shifted value.

```
(x & ~mask) | ((y << shift) & mask);
```

Dangerous example (no longer)

`shift = 7`

`x`	1000110101101101
`y`	0000000000100011
`mask`	0000011110000000
`x & ~mask`	1000100001101101
`((y << shift) & mask)`	0000000110000000
`(x & ~mask) | ((y << shift) & mask)`	1000100111101101