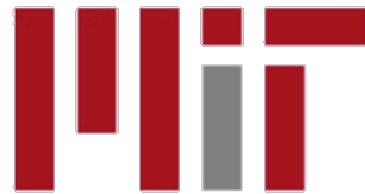
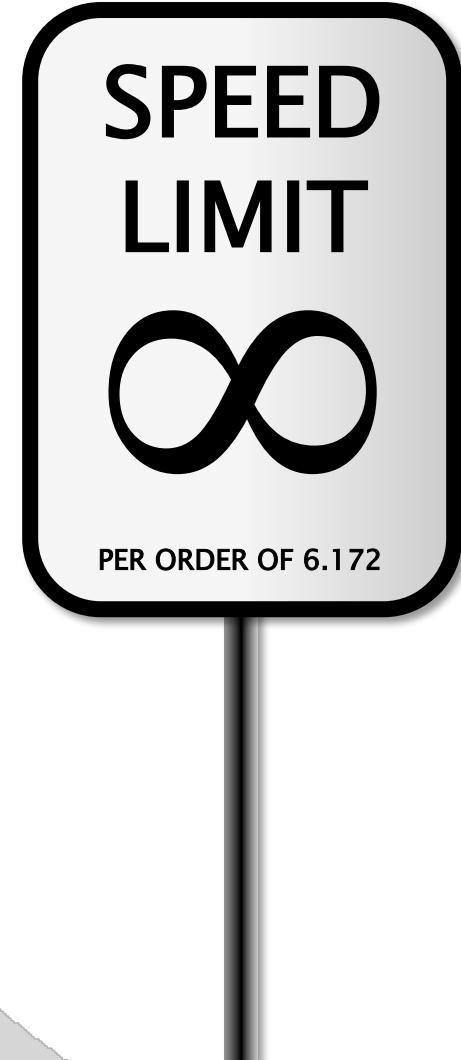


6.172 Performance Engineering of Software Systems



LECTURE 3 Bit Hacks

Charles E. Leiserson
September 11, 2014



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
$x \& mask >> shift$	000000000001010

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

Example

$\text{shift} = 7$

x	1011110101101101
y	0000000000000011
mask	0000011110000000
$x \& \sim\text{mask}$	1011100001101101
$x = (x \& \sim\text{mask}) (y << \text{shift});$	1011100111101101

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 y value.

For safety's sake:
 $((y \ll \text{shift}) \& \text{mask})$

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

Example

$\text{shift} = 7$

x	1011110101101101
y	0000000000000011
mask	0000011110000000
$x \& \sim\text{mask}$	1011100001101101
$x = (x \& \sim\text{mask}) (y \ll \text{shift});$	1011100111101101

Ordinary Swap

Problem

Swap two integers x and y .

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

No-Temp Swap

Problem

Swap x and y without using a temporary.

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

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

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;
```

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

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;
```

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

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;
```

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

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	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse: $(x \wedge y) \wedge y \Rightarrow x$

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

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

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Example

x	10111101			
y	00101110			

Why it works

XOR is its own inverse: $(x \wedge y) \wedge y \Rightarrow x$

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;
```

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Example

x	10111101	10010011		
y	00101110	00101110		

Why it works

XOR is its own inverse: $(x \wedge y) \wedge y \Rightarrow x$

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;
```

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Example

x	10111101	10010011	10010011	
y	00101110	00101110	10111101	

Why it works

XOR is its own inverse: $(x \wedge y) \wedge y \Rightarrow x$

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;
```

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse: $(x \wedge y) \wedge y \Rightarrow x$

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

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

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse: $(x \wedge y) \wedge y \Rightarrow x$

Performance

Poor at exploiting *instruction-level parallelism (ILP)*.

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 may be smart enough to optimize away the unpredictable branch, but maybe not.

No-Branch Minimum

Problem

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

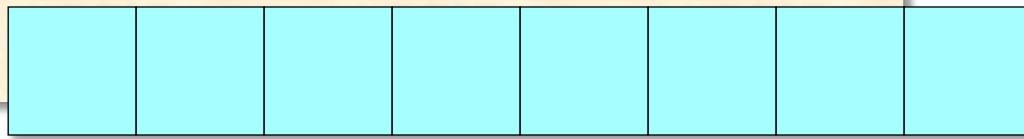
```
r = y ^ ((x ^ y) & -(x < y));
```

Why it works:

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

Merging Two Sorted Arrays

```
static void merge(long * __restrict C,
                  long * __restrict A,
                  long * __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(long * __restrict C,
                  long * __restrict A,
                  long * __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--;
    }
}
```

This optimization works well on some machines, but on modern machines using `gcc -O3`, the branchless version is usually slower than the branching version.  Modern compilers can perform this optimization better than you can!

Why Learn Bit Hacks?

Why learn bit hacks if they don't even work?

- Because the compiler does them, and it will help to understand what the compiler is doing when you look at the assembly language.
- Because sometimes the compiler doesn't optimize, and you have to do it yourself by hand.
- Because many bit hacks for words extend naturally to bit and word hacks for vectors.
- Because these tricks arise in other domains, and so it pays to be educated about them.
- Because they're fun!

Modular Addition

Problem

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

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

Division is expensive, unless by a power of 2.

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

Unpredictable branch is expensive.

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

Same trick as minimum.

Round up to a Power of 2

Problem

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

Notation

$$\lg n = \log_2 n$$

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
0011110001111111

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
0011110001111111
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
0011110001111111
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
0011110001111111
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
0011110001111111
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
0011110001111111
0011111111111111
0100000000000000

Round up to a Power of 2

Problem

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

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

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

Example

001000001010000
001000001001111
0011000001101111
00111100011111111
00111111111111111
01000000000000000

Set bit $\lceil \lg n \rceil$

Populate all bits
to the right with 1

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
0011110001111111
0011111111111111
0100000000000000

Why decrement?

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

Least-Significant 1

Problem

Compute the mask of the least-significant 1 in word x .

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

Example

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

Why it works

The binary representation of $-x$ is $\sim x - 1$.

Question

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

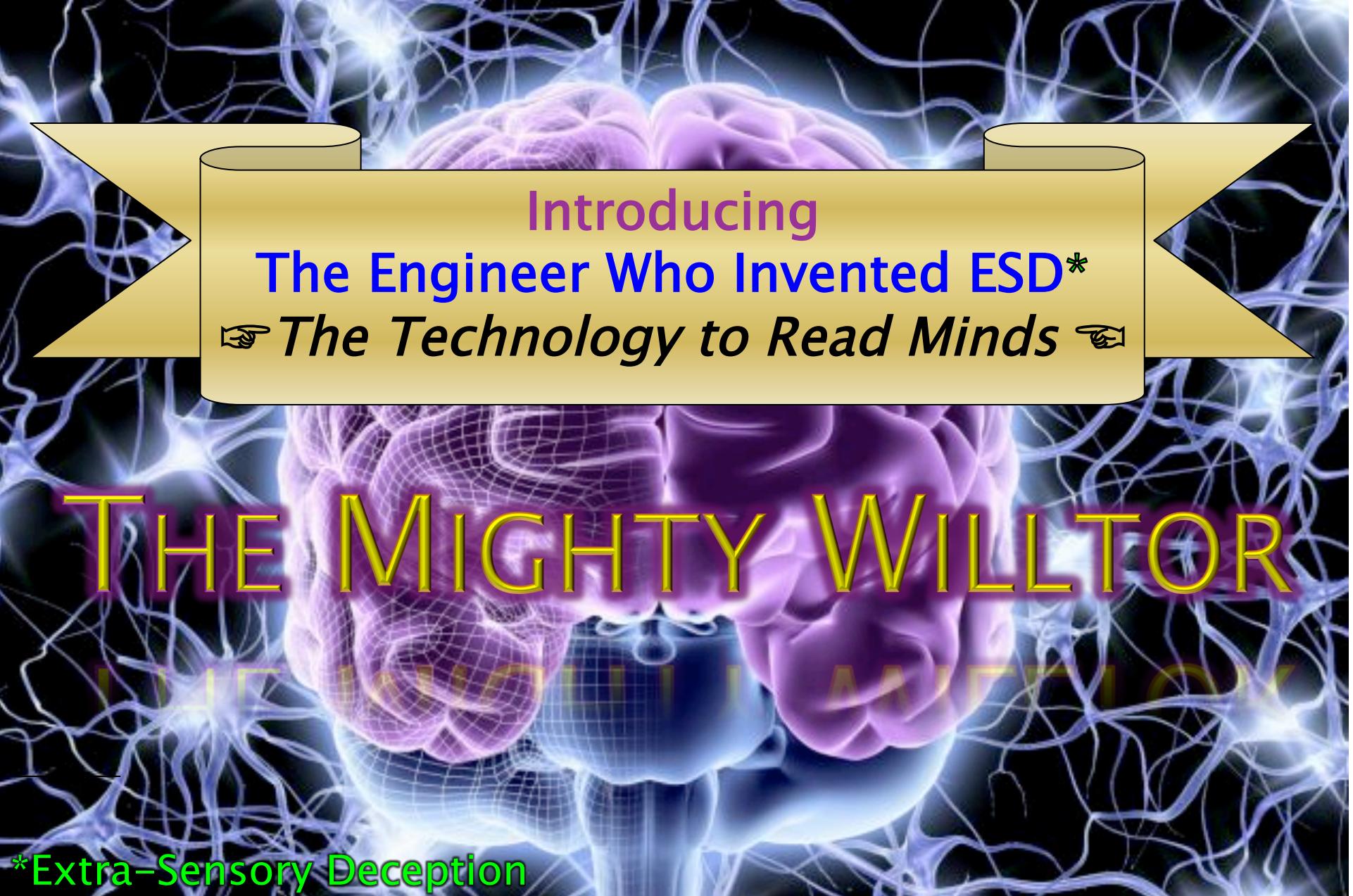
Log Base 2 of a Power of 2

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];
```

Mathemagic Trick



Introducing
The Engineer Who Invented ESD*
☞ *The Technology to Read Minds* ☚

THE MIGHTY WILLTOR

*Extra-Sensory Deception

Log Base 2 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 .

$$\begin{aligned}00011101_2 * 2^4 &\Rightarrow 11010000 \\11010000_2 \gg 5 &\Rightarrow 6 \\ \text{convert}[6] &\Rightarrow 4\end{aligned}$$

start with
all 0's

Example: $k=3$

	00011101 ₂
0	000
1	001
2	011
3	111
4	110
5	101
6	010
7	100

Performance

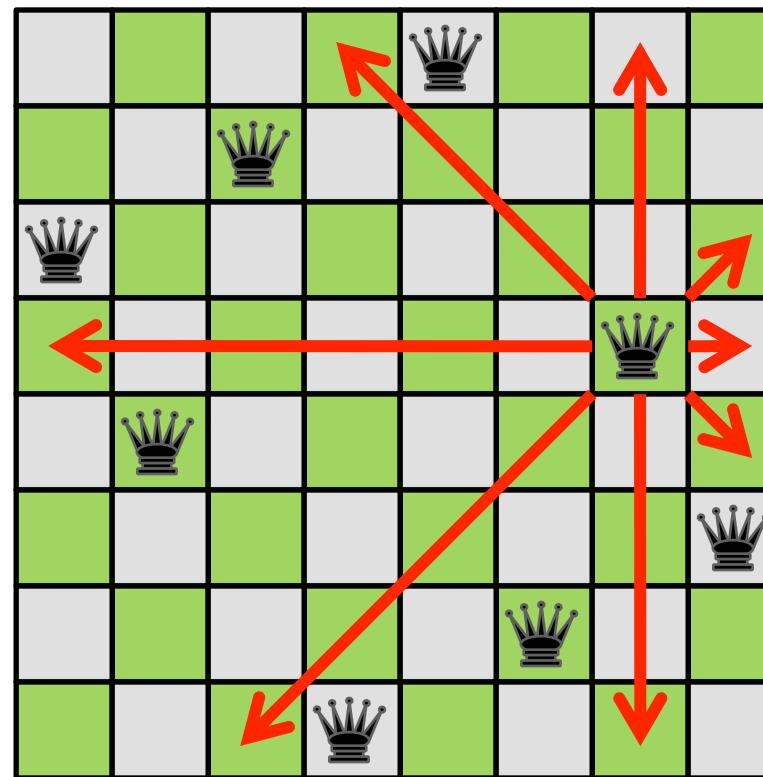
Limited by multiply and table look-up.

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

Queens Problem

Problem

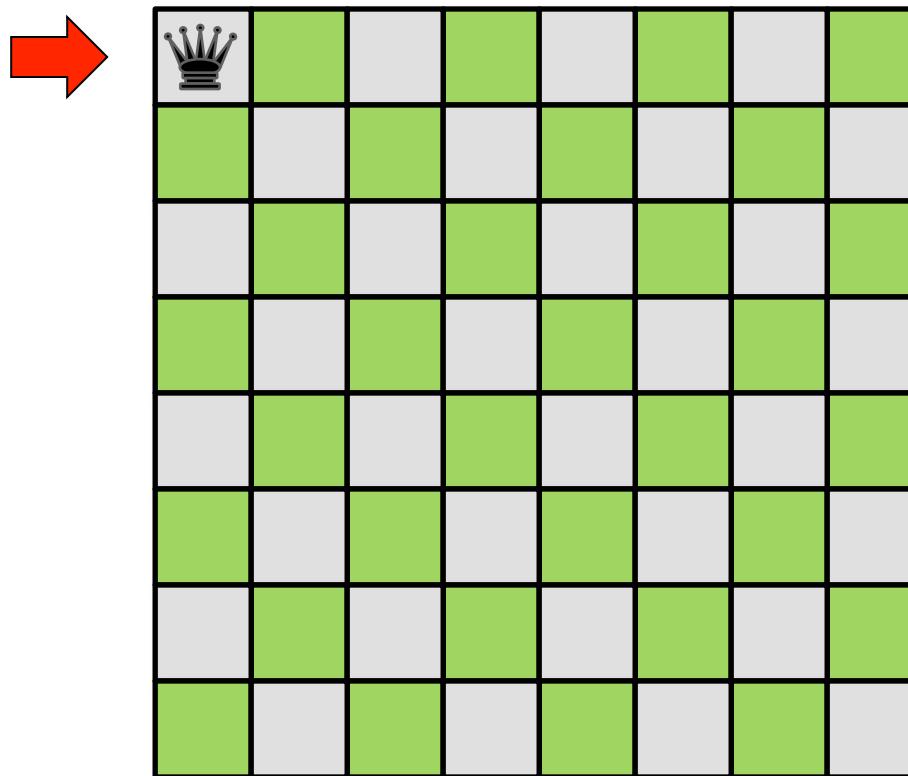
Place n queens on an $n \times n$ chessboard so that no queen attacks another, i.e., no two queens in any row, column, or diagonal.



Backtracking Search

Strategy

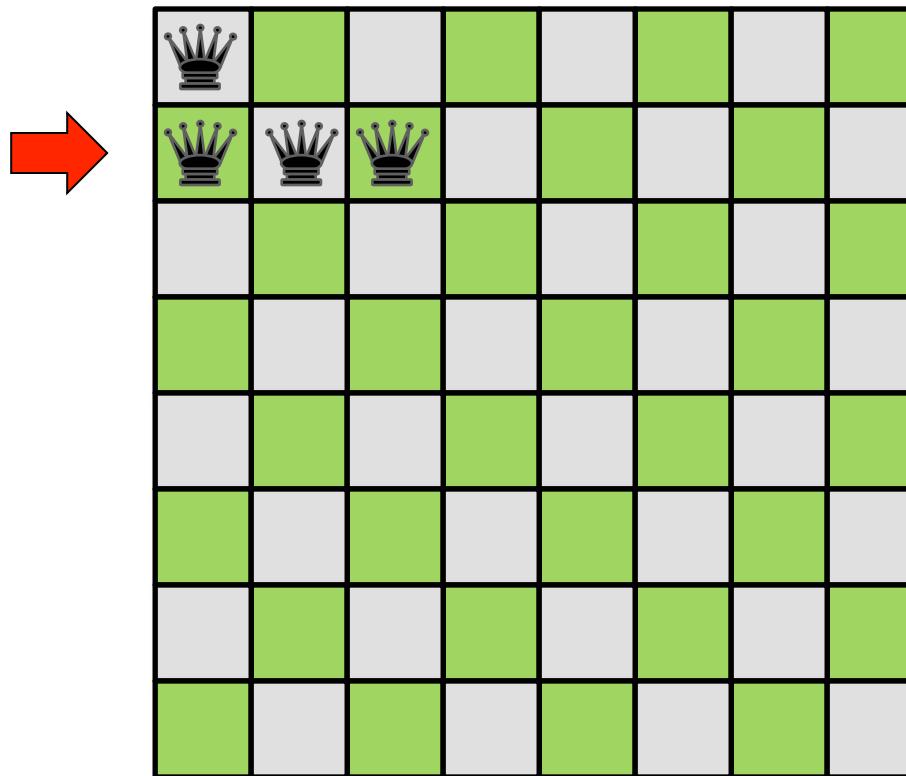
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

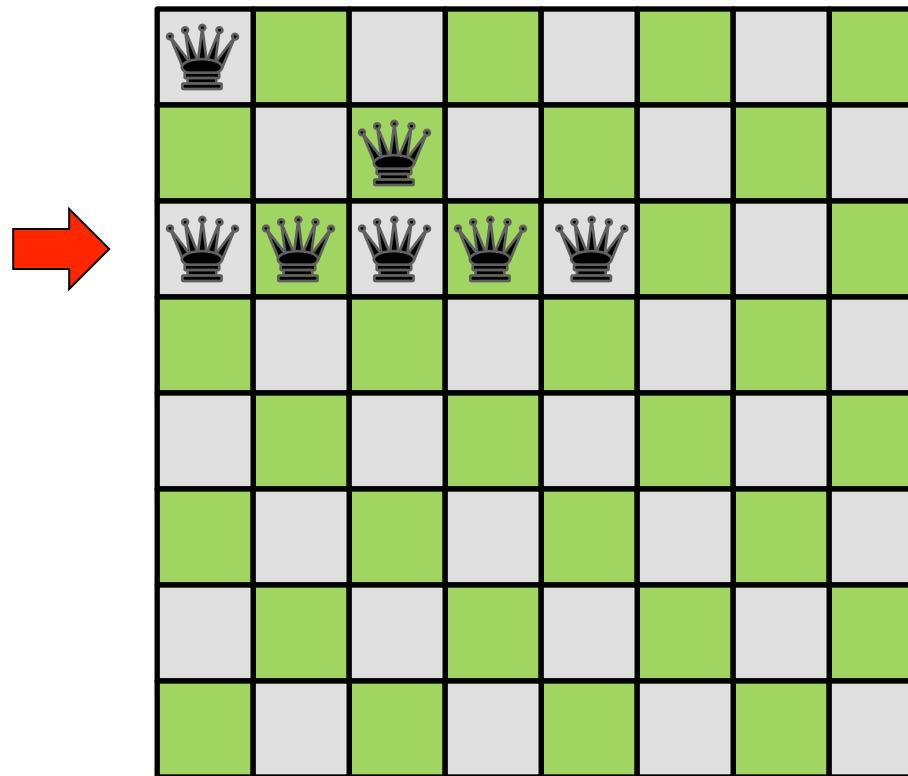
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

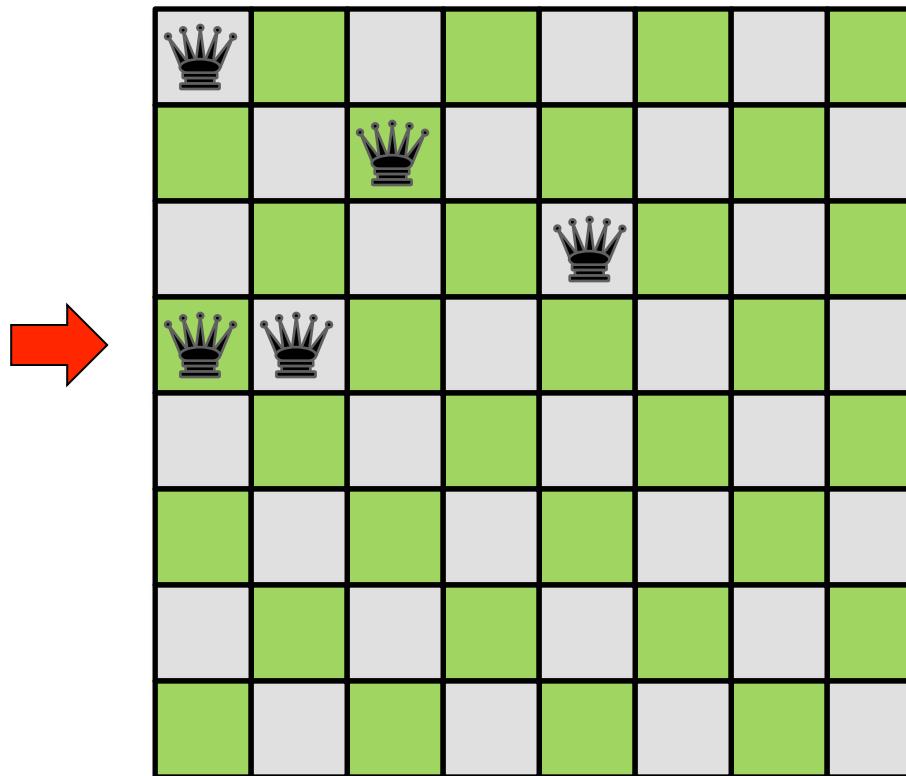
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

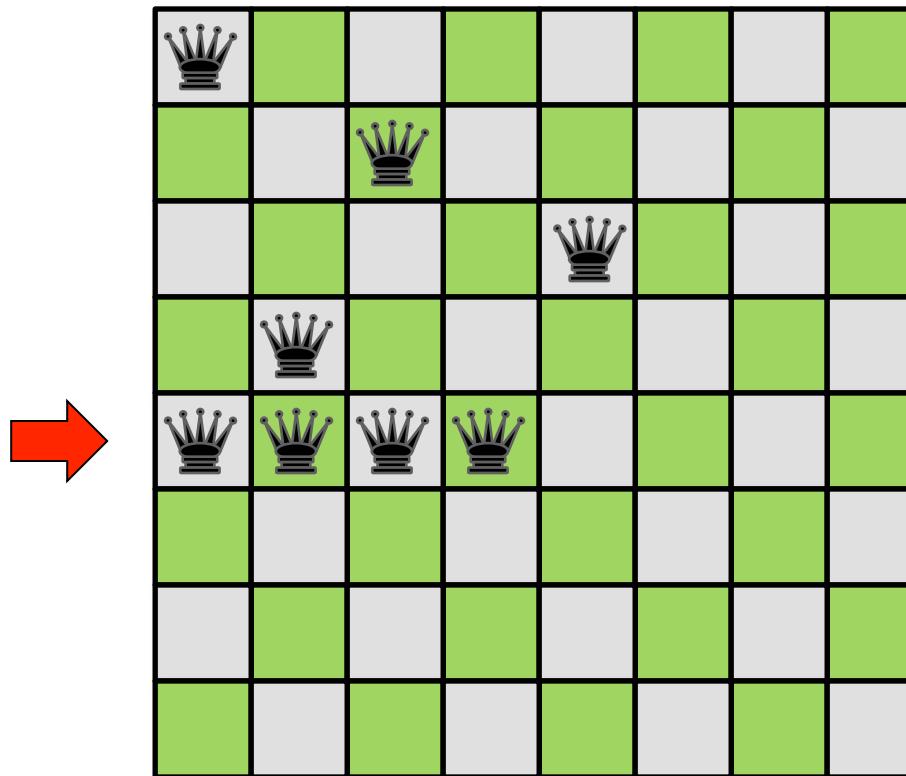
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

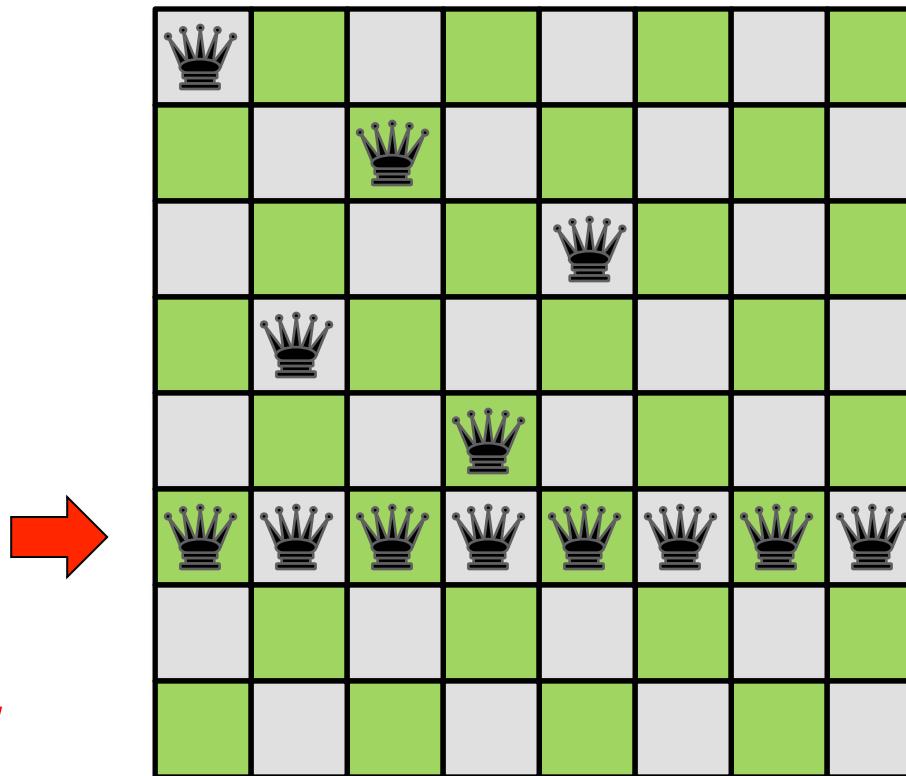
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

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

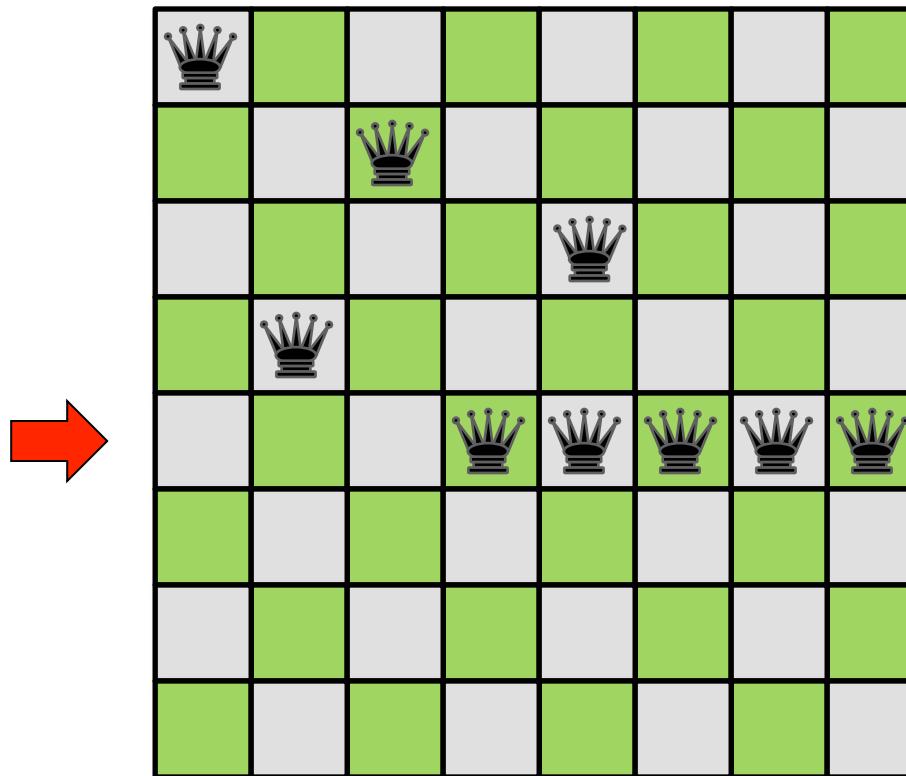


Backtrack!

Backtracking Search

Strategy

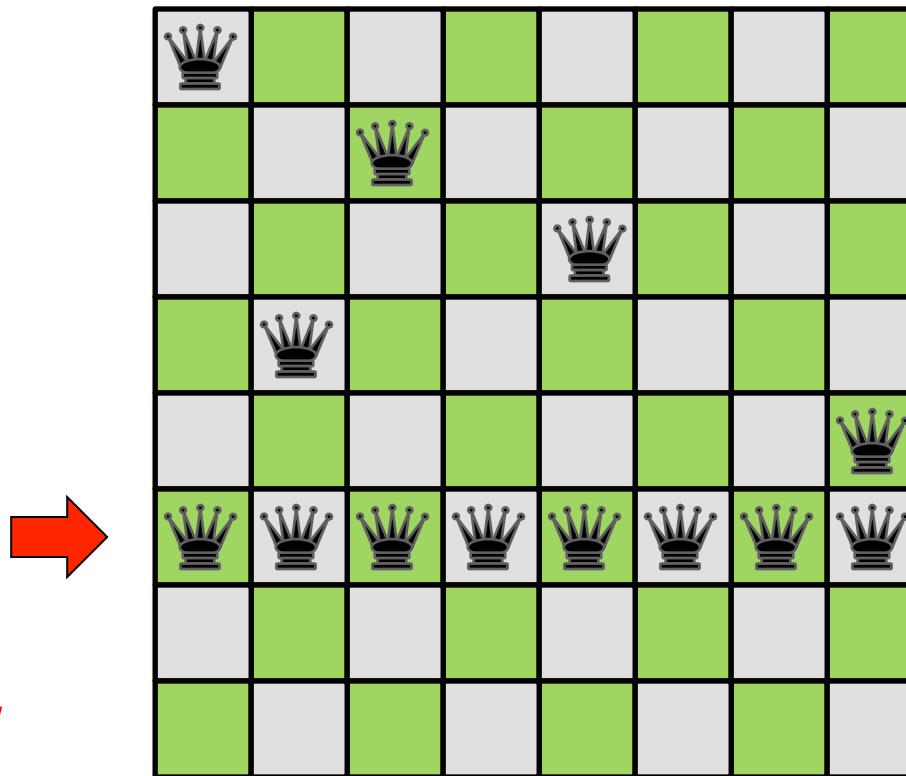
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

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

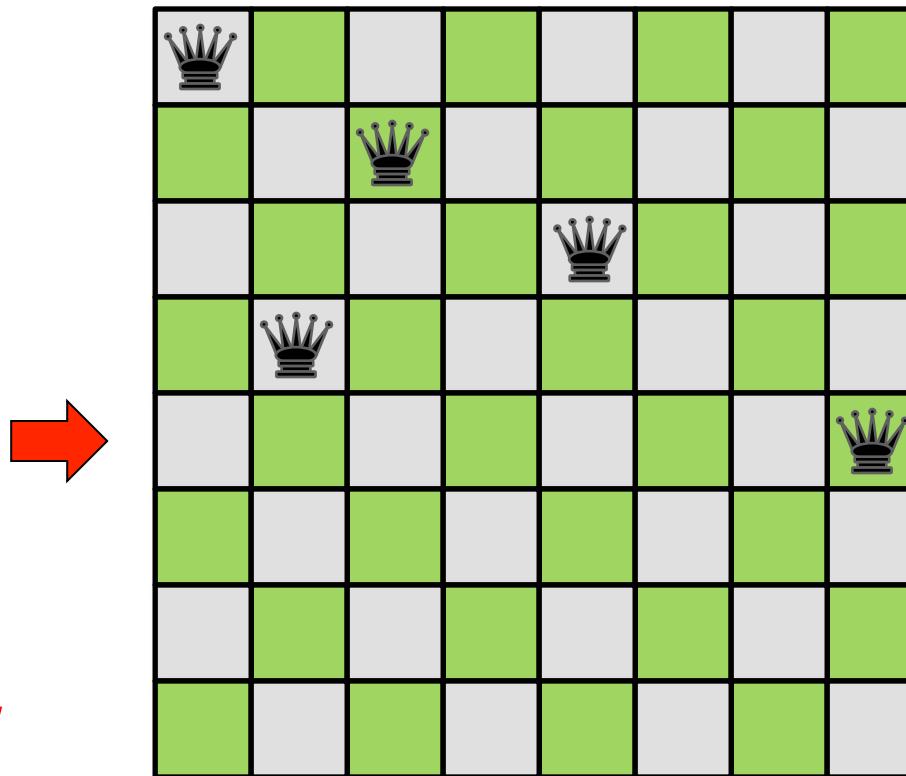


Backtrack!

Backtracking Search

Strategy

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

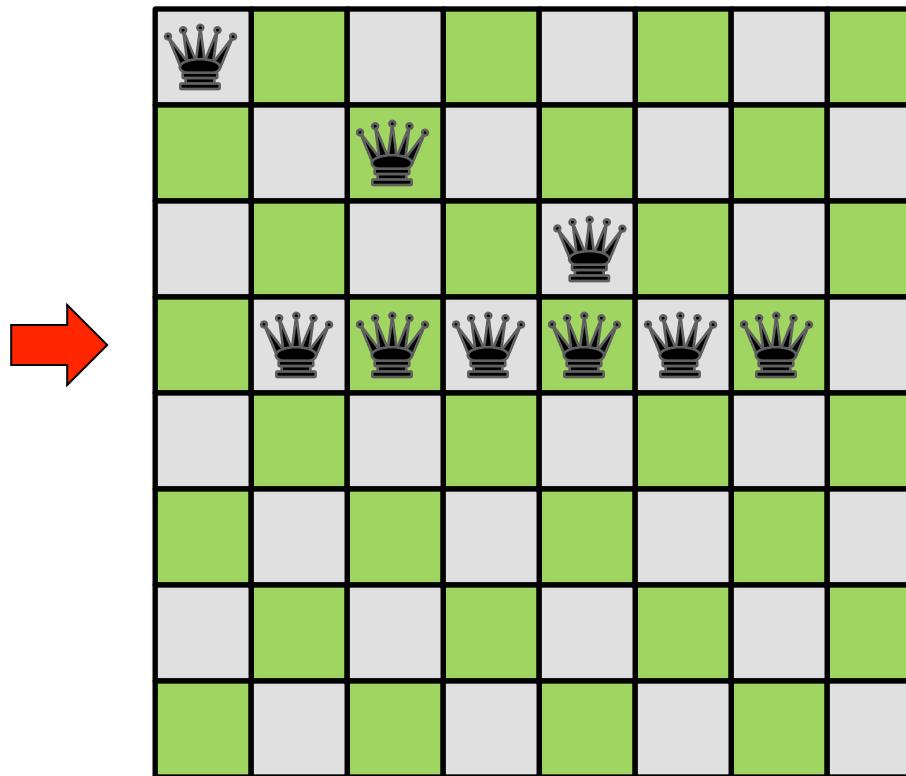


Backtrack!

Backtracking Search

Strategy

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

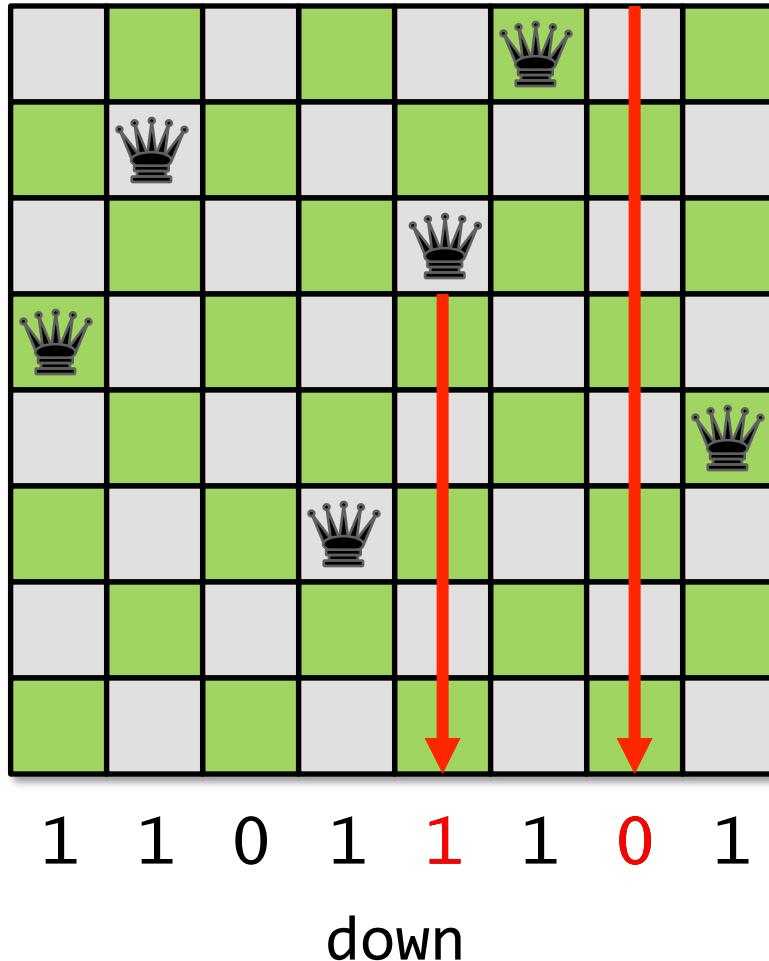


Board Representation

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

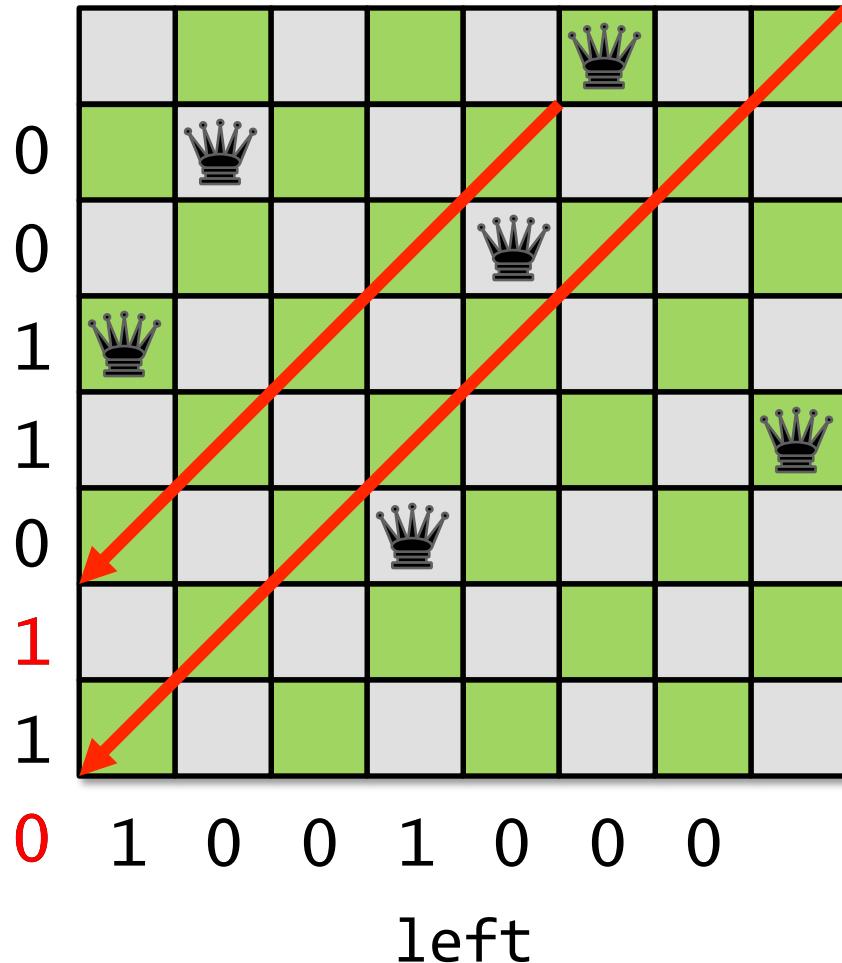
- array of n^2 bytes?
- array of n^2 bits?
- array of n bytes?
- 3 bitvectors of size n , $2n-1$, and $2n-1!$

Bitvector Representation



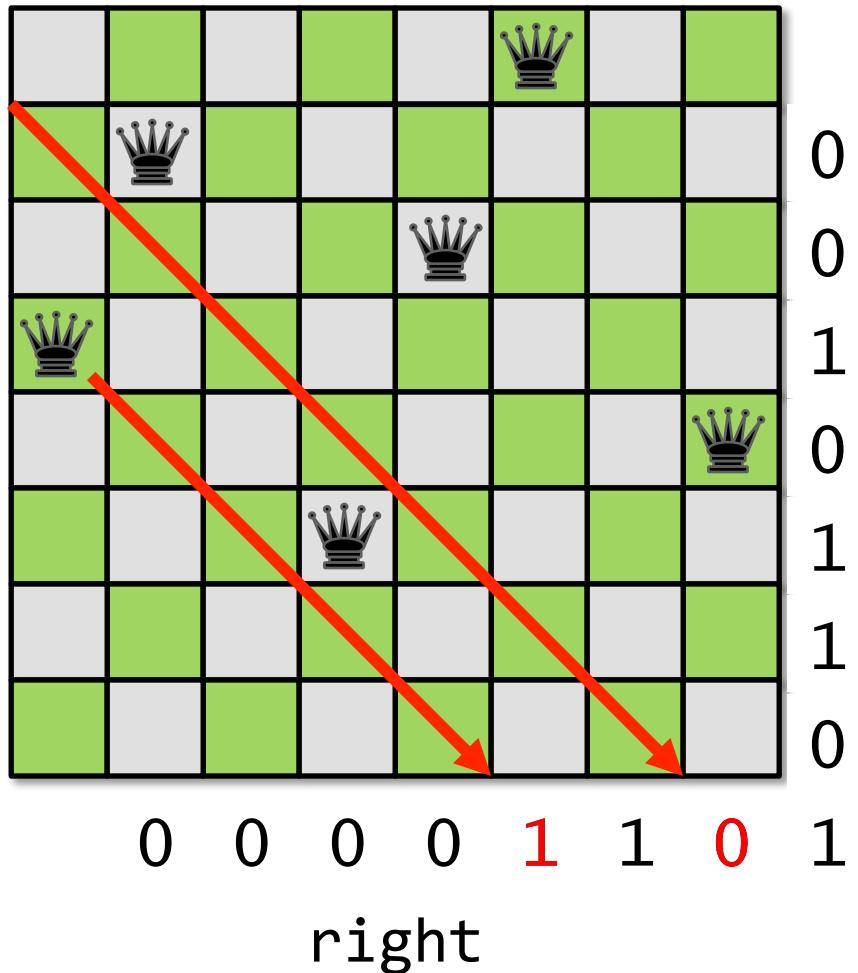
Placing a queen in column c is not safe if
down & $(1 \ll c)$;
is nonzero.

Bitvector Representation



Placing a queen in row r and column c is not safe if
 $\text{left} \& (1 << (r+c))$ is nonzero.

Bitvector Representation



Placing a queen in row r and column c is not safe if
right & ($1 \ll (n-r+c)$)
is nonzero.

Population Count I

Problem

Count the number of **1** bits in a word **x**.

```
for (r=0; x!=0; ++r)  
    x &= x - 1;
```

Repeatedly eliminate
the least-significant 1.

Example

x	00101101110 1 0000
x - 1	00101101110 0 1111
x & (x - 1);	00101101110 0 00000

Issue

Fast if the popcount is small, but in the worst case, the running time is proportional to the number of bits in the word.

Population Count II

Table look-up

```
static const int count[256] =  
{ 0, 1, 1, 2, 1, 2, 2, 3, 1, ..., 8 };  
  
for (int r = 0; x != 0; x >>= 8)  
    r += count[x & 0xFF];
```

Performance

Performance depends on the size of x . The cost of memory operations is a major bottleneck.

- register: 1 cycle,
 - L1-cache: 4 cycles,
 - L2-cache: 10 cycles,
 - L3-cache: 50 cycles,
 - DRAM: 150 cycles.
- } per 64-byte cache line

Population Count III

Parallel divide-and-conquer

```
// Create masks
M5 = ~((-1) << 32);      // 032132
M4 = M5 ^ (M5 << 16);    // (016116)2
M3 = M4 ^ (M4 << 8);     // (0818)4
M2 = M3 ^ (M3 << 4);     // (0414)8
M1 = M2 ^ (M2 << 2);     // (0212)16
M0 = M1 ^ (M1 << 1);     // (01)32

// Compute popcount
x = ((x >> 1) & M0) + (x & M0);
x = ((x >> 2) & M1) + (x & M1);
x = ((x >> 4) + x) & M2;
x = ((x >> 8) + x) & M3;
x = ((x >> 16) + x) & M4;
x = ((x >> 32) + x) & M5;
```

Notation:

$$X^k = \underbrace{XX \cdots X}_{k \text{ times}}$$

Population Count III

1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 0 1 0 0 0 1 1 1 0 0 0

Population Count III

```
1100001001011011111010001111000
```

$x \& M0$

$(x >> 1) \& M0$

Population Count III

1	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	0	1	0	0	0	1	1	1	1	0	0
+	1	1	0	0	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1	0	0
	1	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	1	1	0	0	1	0	0	

Population Count III

$$\begin{array}{r} 1100001001011011111010001111000 \\ 10000001010101101010010001100100 \\ + 10000001010101101010010001100100 \\ \hline 10000000101010110101010010001100100 \end{array}$$

$x \& M0$
 $(x >> 1) \& M0$
 $x \& M1$
 $(x >> 2) \& M1$

Population Count III

$$\begin{array}{r} 1100001001011011111010001111000 \\ + \quad 10001101111110011100 \\ \hline 00010000100100011010000100110001 \end{array}$$

$x \& M_0$
 $(x \gg 1) \& M_0$
 $x \& M_1$
 $(x \gg 2) \& M_1$

Population Count III

$$\begin{array}{r} 1100001001011011111010001111000 \\ + \quad 10001101111110011100 \\ \hline 000100001001000110100000100110001 \end{array}$$

$x \& M_0$
 $(x \gg 1) \& M_0$
 $x \& M_1$
 $(x \gg 2) \& M_1$
 $x \& M_2$
 $(x \gg 4) \& M_2$

Population Count III

$$\begin{array}{r}
 1100001001011011111010001111000 \\
 + 10000110110111110011100 \\
 \hline
 00010101101000101000100
 \end{array}
 \quad \text{x\&M0} \quad (x>>1)\&M0$$

$$\begin{array}{r}
 + 1000010101100101010101 \\
 \hline
 0001001100100100100101
 \end{array}
 \quad \text{x\&M1} \quad (x>>2)\&M1$$

$$\begin{array}{r}
 + 0010001001001001001001 \\
 \hline
 00000011000000110000101000100
 \end{array}
 \quad \text{x\&M2} \quad (x>>4)\&M2$$

Population Count III

	1	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	0	1	0	0	0	1	1	1	1	0	0
+	1	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	x&M0 (x>>1)&M0		
	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0	0	0	x&M1 (x>>2)&M1		
+	1	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	x&M2 (x>>4)&M2		
	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	x&M4 (x>>8)&M4		

Population Count III

$$\begin{array}{r}
 110000100101101111101000111000 \\
 + 100001101111110011100 \\
 \hline
 0001001011010001001000
 \end{array}
 \quad \begin{array}{l} x\&M0 \\ (x>>1)\&M0 \end{array}$$

$$\begin{array}{r}
 + 10000101011001010101 \\
 \hline
 0001001011010001001000
 \end{array}
 \quad \begin{array}{l} x\&M1 \\ (x>>2)\&M1 \end{array}$$

$$\begin{array}{r}
 + 0010001001010001001000 \\
 \hline
 0000010100000101000100
 \end{array}
 \quad \begin{array}{l} x\&M2 \\ (x>>4)\&M2 \end{array}$$

$$\begin{array}{r}
 + 0000001100000101000100 \\
 \hline
 0000000000000100100100
 \end{array}
 \quad \begin{array}{l} x\&M4 \\ (x>>8)\&M4 \end{array}$$

Population Count III

$$\begin{array}{r}
 1100001001011011111010001111000 \\
 + 1000110111110011100011100 \\
 \hline
 000100101101000100100100
 \end{array} \quad \begin{array}{l} x \& M0 \\ (x >> 1) \& M0 \end{array}$$

$$\begin{array}{r}
 + 1000100101100101010101 \\
 \hline
 0001001001010100101100
 \end{array} \quad \begin{array}{l} x \& M1 \\ (x >> 2) \& M1 \end{array}$$

$$\begin{array}{r}
 + 0010001001010100101100 \\
 \hline
 0000010100000100
 \end{array} \quad \begin{array}{l} x \& M2 \\ (x >> 4) \& M2 \end{array}$$

$$\begin{array}{r}
 + 0000001100000101 \\
 \hline
 00000000000000001001
 \end{array} \quad \begin{array}{l} x \& M3 \\ (x >> 8) \& M3 \end{array}$$

$$\begin{array}{r}
 + 00000000000000001001 \\
 \hline
 000000000000000000001001
 \end{array} \quad \begin{array}{l} x \& M4 \\ (x >> 16) \& M4 \end{array}$$

Population Count III

1	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	0	1	0	0	0	1	1	1	0	0	
+	1	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	x&M0		
+	1	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	(x>>1)&M0		
	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	x&M1		
+	1	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	(x>>2)&M1		
	0	0	0	1	0	0	1	1	0	1	1	0	0	0	0	0	1	0	0	0	1	0	0	0	x&M2		
+	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	(x>>4)&M2		
	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	x&M3		
+	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	(x>>8)&M3		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	x&M4		
+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(x>>16)&M4		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Population Count III

Parallel divide-and-conquer

```
// Create masks
M5 = ~((-1) << 32);      // 032132
M4 = M5 ^ (M5 << 16);    // (016116)2
M3 = M4 ^ (M4 << 8);     // (0818)4
M2 = M3 ^ (M3 << 4);     // (0414)8
M1 = M2 ^ (M2 << 2);     // (0212)16
M0 = M1 ^ (M1 << 1);     // (01)32

// Compute popcount
x = ((x >> 1) & M0) + (x & M0);
x = ((x >> 2) & M1) + (x & M1);
x = ((x >> 4) + x) & M2;
x = ((x >> 8) + x) & M3;
x = ((x >> 16) + x) & M4;
x = ((x >> 32) + x) & M5;
```

Performance
 $\Theta(\lg w)$ time,
where w =
word length.

Avoid
overflow

No worry
about
overflow.

Popcount Instructions

Most modern machines provide **popcount** instructions, which operate much faster than anything you can code yourself. You can access them via compiler intrinsics, e.g., in GCC:

```
int __builtin_popcount (unsigned int x);
```

Warning: You may need to enable certain compiler switches to access built-in functions, and your code will be less portable.

Exercise

Compute the log base **2** of a power of **2** quickly using a **popcount** instruction.

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.

Happy Bit-Hacking!