Problem
Swap two integers $x$ and $y$.

t = x;
x = y;
y = t;
No-Temp Swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

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

Example

<table>
<thead>
<tr>
<th>$x$</th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>
No-Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
    x &= x \land y; \\
    y &= x \land y; \\
    x &= x \land y;
\end{align*}
\]

Example

<table>
<thead>
<tr>
<th></th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10111101</td>
<td>10010011</td>
<td>10010011</td>
<td>00101110</td>
</tr>
<tr>
<td>y</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>
No-Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
x &= x \, \text{^} \, y; \\
y &= x \, \text{^} \, y; \\
x &= x \, \text{^} \, y;
\end{align*}
\]

Example
\[
\begin{array}{cccc}
x & 10111101 & 10010011 & 10010011 & 00101110 \\
y & 00101110 & 00101110 & 10111101 & 10111101 \\
\end{array}
\]
No-Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
x &= x \text{^} y; \\
y &= x \text{^} y; \\
x &= x \text{^} y;
\end{align*}
\]

Example
\[
\begin{array}{c|c|c|c|c}
\hline
x & 10111101 & 10010011 & 10010011 & 00101110 \\
y & 00101110 & 00101110 & 10111101 & 10111101 \\
\hline
\end{array}
\]
No-Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
x = x \oplus y;
\]
\[
y = x \oplus y;
\]
\[
x = x \oplus y;
\]

Example
\[
\begin{array}{cccc}
x & 10111101 & 10010011 & 10010011 & 00101110 \\
y & 00101110 & 00101110 & 10111101 & 10111101 \\
\end{array}
\]

Why it works
XOR is its own inverse: \((x \oplus y) \oplus y = x\).

Performance
Poor at exploiting instruction-level parallelism (ILP).
**Problem**
Find the minimum \( r \) of two integers \( x \) and \( y \).

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

or

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

**Performance**
A mispredicted branch empties the processor pipeline
- \(~16\) cycles on the cloud facility’s Intel Core i7’s.
The compiler might be smart enough to avoid the unpredictable branch, but maybe not.
No-Branch Minimum

Problem
Find the minimum \( z \) of two integers \( x \) and \( y \) without a branch.

\[
r = y \land ((x \land y) \land \lnot(x < y));
\]

Why it works:
• C represents the Booleans TRUE and FALSE with the integers 1 and 0, respectively.
• If \( x < y \), then \( \lnot(x < y) = -1 \), which is all 1’s in two’s complement representation. Therefore, we have \( y \land (x \land y) = x \).
• If \( x \geq y \), then \( \lnot(x < y) = 0 \). Therefore, we have \( y \land 0 = y \).
Modular Addition

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

- \(r = (x + y) \% n;\)
- Divide 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 $2^{\lceil \log n \rceil}$.

//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;

Example

<table>
<thead>
<tr>
<th></th>
<th>00100000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0010000001001111</td>
</tr>
<tr>
<td></td>
<td>0011000001101111</td>
</tr>
<tr>
<td></td>
<td>0011110001111111</td>
</tr>
<tr>
<td></td>
<td>0011111111111111</td>
</tr>
<tr>
<td></td>
<td>0100000000000000</td>
</tr>
</tbody>
</table>
Problem
Compute $2^{\lceil \log n \rceil}$.

```
//64-bit integers
--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
```
Problem
Compute \(2^{\lceil \log n \rceil}\).

//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;

Example

\[
\begin{array}{c}
0010000001010000 \\
0010000001001111 \\
0011000001101111 \\
0011110001111111 \\
0011111111111111 \\
0100000000000000
\end{array}
\]
Problem
Compute $2^\lceil \log n \rceil$.

// 64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;

Example

<table>
<thead>
<tr>
<th>0010000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001001111</td>
</tr>
<tr>
<td>0011000001101111</td>
</tr>
<tr>
<td>0011110001111111</td>
</tr>
<tr>
<td>0011111000111111</td>
</tr>
<tr>
<td>0100000000000000</td>
</tr>
</tbody>
</table>
Round up to a Power of 2

Problem
Compute $2^\lceil \log n \rceil$.

```
//64-bit integers
--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 $2^\lceil \log n \rceil$.

//64-bit integers
--n;
n | = n >> 1;
n | = n >> 2;
n | = n >> 4;
n | = n >> 8;
n | = n >> 16;
n | = n >> 32;
++n;

Example

| 00100000001010000 |
| 00100000001001111 |
| 00110000001101111 |
| 00111100001111111 |
| 00111111111111111 |
| 01000000000000000 |
Round up to a Power of 2

Problem
Compute \(2^{\lceil \log n \rceil}\).

```
// 64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

<table>
<thead>
<tr>
<th>n</th>
<th>Incremented n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001010000</td>
<td>0010000001001111</td>
</tr>
<tr>
<td>0010000001001111</td>
<td>0011000001101111</td>
</tr>
<tr>
<td>0011000001101111</td>
<td>0011110001111111</td>
</tr>
<tr>
<td>0011110001111111</td>
<td>0011111111111111</td>
</tr>
<tr>
<td>0011111111111111</td>
<td>0100000000000000</td>
</tr>
</tbody>
</table>

Why decrement and increment?
To handle the boundary case when \(n\) is a power of 2.
Problem
Compute the mask of the least-significant 1 in word $x$.

$$r = x \& (-x);$$

Example

<table>
<thead>
<tr>
<th>$x$</th>
<th>00100000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-x$</td>
<td>11011111110110000</td>
</tr>
<tr>
<td>$x &amp; (-x)$</td>
<td>00000000000010000</td>
</tr>
</tbody>
</table>

Question
How do you find the index of the bit, i.e., $\lg r = \log_2 r$?
Problem
Compute \( \log_2 x \), where \( x \) is a power of 2.

```c
const uint64_t deBruijn = 0x022fdd63cc95386d;
const unsigned 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];
```
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**  
$k = 3$

$$
\begin{array}{c}
00011101_2 \\
0 000 \\
1 001 \\
2 011 \\
3 111 \\
4 110 \\
5 101 \\
6 010 \\
7 100 \\
\end{array}
$$

$$00011101_2 \times 2^4 = 11010000_2$$

$$11010000_2 \gg 5 = 6$$

`convert[6] = 4`

Performance

Limited by multiply and table look-up

```text
convert[8] = {0, 1, 6, 2, 7, 5, 4, 3};
```
Population Count I

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

\[
\text{for (r=0; x!=0; ++r)
\quad x &= x - 1;}
\]
Repeatedly eliminate the least-significant 1.

Example

<table>
<thead>
<tr>
<th>x</th>
<th>0010110111010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-1</td>
<td>0010110111001111</td>
</tr>
<tr>
<td>x &amp; (x-1)</td>
<td>0010110111000000</td>
</tr>
</tbody>
</table>

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

```c
static const int count[256] =
{0,1,1,2,1,2,2,3,1,...,8}; // #1's in index

for (r=0; x!=0; x>>=8)
  r += count[x & 0xFF];
```

Performance
Memory operations are much more costly than register operations:
- register: 1 cycle (6 ops issued per cycle per core),
- 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

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

// Create masks
B5 = !((-1) << 32);
B4 = B5 ^ (B5 << 16);
B3 = B4 ^ (B4 << 8);
B2 = B3 ^ (B3 << 4);
B1 = B2 ^ (B2 << 2);
B0 = B1 ^ (B1 << 1);

// Compute popcount
x = ((x >> 1) & B0) + (x & B0);
x = ((x >> 2) & B1) + (x & B1);
x = ((x >> 4) + x) & B2;
x = ((x >> 8) + x) & B3;
x = ((x >> 16) + x) & B4;
x = ((x >> 32) + x) & B5;
Population Count III

11110101000110000011011111001010
11110101000110000011011111001010
Population Count III

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
+ & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]
Population Count III

\[ \begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
+ & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{array} \]
### Population Count III

<table>
<thead>
<tr>
<th>1 1 1 0 1 0 0 0 1 1</th>
<th>0 1 0 1 1 1 1 0 0 1 0 1 1 0</th>
<th>1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 1 1 0 1 1</td>
<td>0 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0</td>
</tr>
<tr>
<td></td>
<td>10 01 01 00 10 10 00 01 10 00 01</td>
<td>01 01 01 00 10 00 01 10 00 01</td>
</tr>
<tr>
<td></td>
<td>01 00 01 00 01 00 01 10 00 01</td>
<td>00 10 00 01 10 00 01</td>
</tr>
<tr>
<td></td>
<td>01 00 01 00 01 00 01 10 00 01</td>
<td>00 10 00 01 10 00 01</td>
</tr>
<tr>
<td></td>
<td>01 00 01 00 01 00 01 10 00 01</td>
<td>00 10 00 01 10 00 01</td>
</tr>
<tr>
<td></td>
<td>11110101000110000011011111001010</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 1 0 1 0 0 0 1 1 1 1 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 01 01 01 00 10 10 00 01 10 01</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>10 01 00 01 00 01 10 01 00 01 10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01000010000100010010001100100010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>+</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000001</td>
<td>11000000</td>
</tr>
</tbody>
</table>

Population Count III
### Population Count III

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>10</td>
<td>10</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>10</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---+---+---+---+---+---+---+---+---</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0100</td>
<td>0001</td>
<td>0010</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---+---+---+---+---</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011000000001000000101000000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Population Count III

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>10</td>
<td>10</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>10</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0100</td>
<td>0001</td>
<td>0010</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000000010</td>
<td>000000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>000000110</td>
<td>000000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00000000000001000</td>
<td>00000000000001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Population Count III

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>10</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>10</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0100</td>
<td>0001</td>
<td>0010</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00000010</td>
<td>000000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>000000110</td>
<td>000000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00000001100000001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2009 Charles E. Leiserson
### Population Count III

```
1111010100011000000110111111001010
+ 1 1 1 1 1 0 1 0 0 0 1 1 1 1 1 0 0 0 0
+ 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1
+ 10 01 01 00 10 10 00 01
+ 10 01 00 01 00 01 10 01
+ 0010 0001 0011 0010
+ 0100 0001 0010 0010
+ 00000010 00000100
+ 00000110 00000101
+ 0000000000001001
+ 0000000000001000
0000000000000000000000100001
```

17
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.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
**Strategy**
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
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.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
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 $2n-1$!
Placing a queen in column \( c \) is not safe if 
\[
\text{down} \land (1 << c)
\]
is nonzero.
Bitvector Representation

Placing a queen in row $r$ and column $c$ is not safe if $\text{right} \& (1<<(n-r+c))$ is nonzero.
Placing a queen in row \( r \) and column \( c \) is not safe if
\[
\text{left} \ & \ (1 \ll (r+c))
\]
is nonzero.
Sean Eron Anderson, “Bit twiddling hacks,”

Happy Hacking!