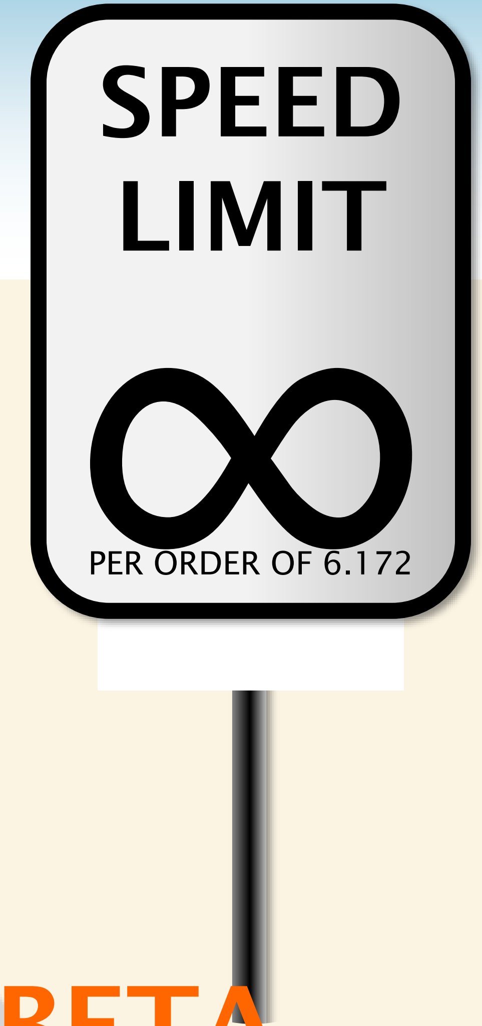


Sit near your project mates!

RECITATION 2
BIT HACKS, PROJECT 1 BETA

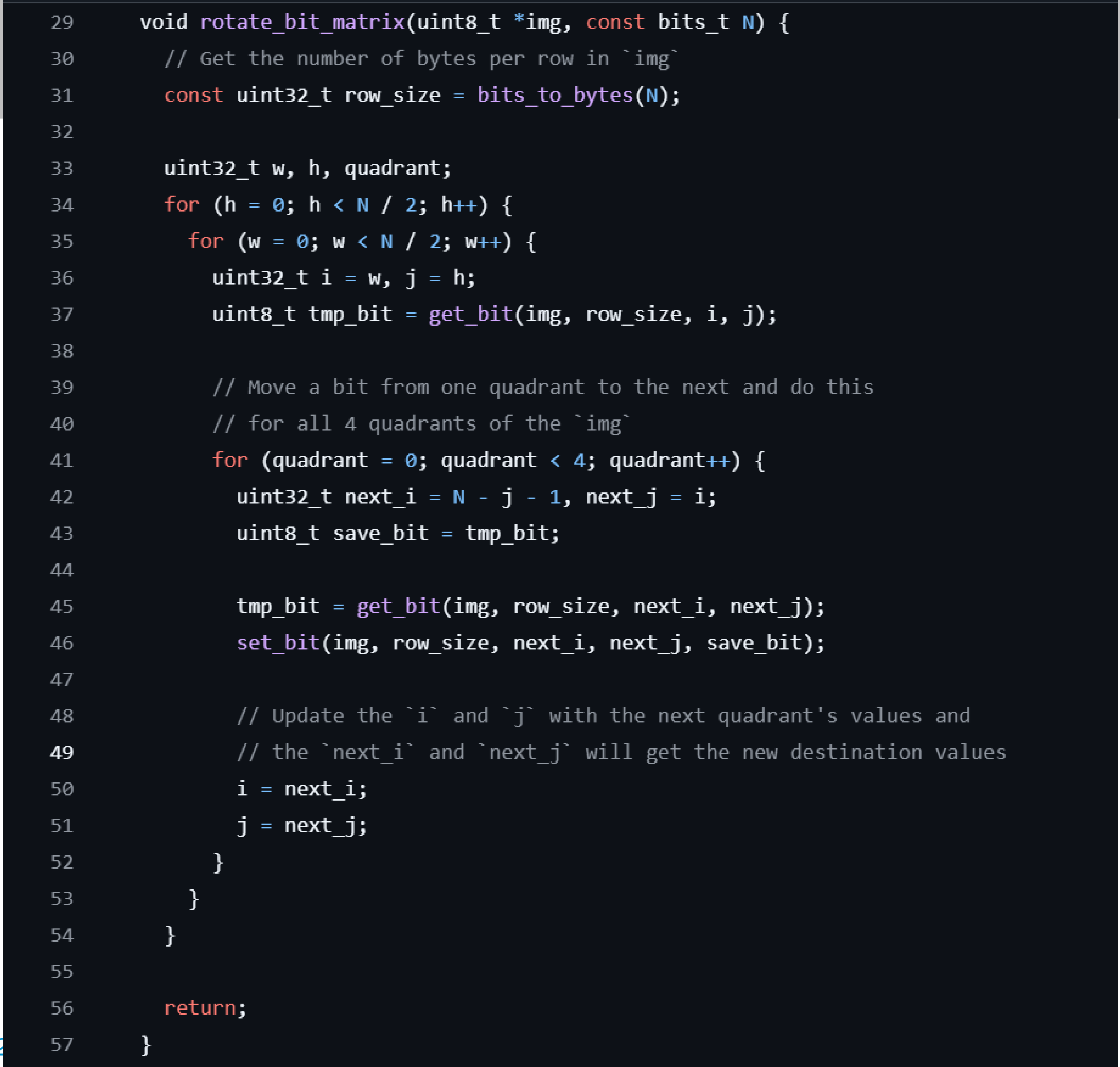
Quick Announcements

Homework:

1. HW1 Due date has been moved to Jan 23
2. HW2 Start and Due dates remain unaffected

Advice on the project:

1. Project takes a while to complete, every attempt needs a correctness check before a performance check.
2. Ensure that you make a few attempts every week to ensure success.



```c
void rotate_bit_matrix(uint8_t *img, const bits_t N) {
  // Get the number of bytes per row in `img`
  const uint32_t row_size = bits_to_bytes(N);

  uint32_t w, h, quadrant;
  for (h = 0; h < N / 2; h++) {
    for (w = 0; w < N / 2; w++) {
      uint32_t i = w, j = h;
      uint8_t tmp_bit = get_bit(img, row_size, i, j);

      // Move a bit from one quadrant to the next and do this
      // for all 4 quadrants of the `img`
      for (quadrant = 0; quadrant < 4; quadrant++) {
        uint32_t next_i = N - j - 1, next_j = i;
        uint8_t save_bit = tmp_bit;

        tmp_bit = get_bit(img, row_size, next_i, next_j);
        set_bit(img, row_size, next_i, next_j, save_bit);

        // Update the `i` and `j` with the next quadrant's values and
        // the `next_i` and `next_j` will get the new destination values
        i = next_i;
        j = next_j;
      }
    }
  }

  return;
}
```

Tips for Bit Manipulation

Tips for Bit Manipulation

- General tips:
  - If manipulating bits, generally want to use unsigned ints
  - If not manipulating bits (arithmetic operations), use signed int.
    - Underflow in unsigned numbers (such as N→0 for loop) doesn't work nicely

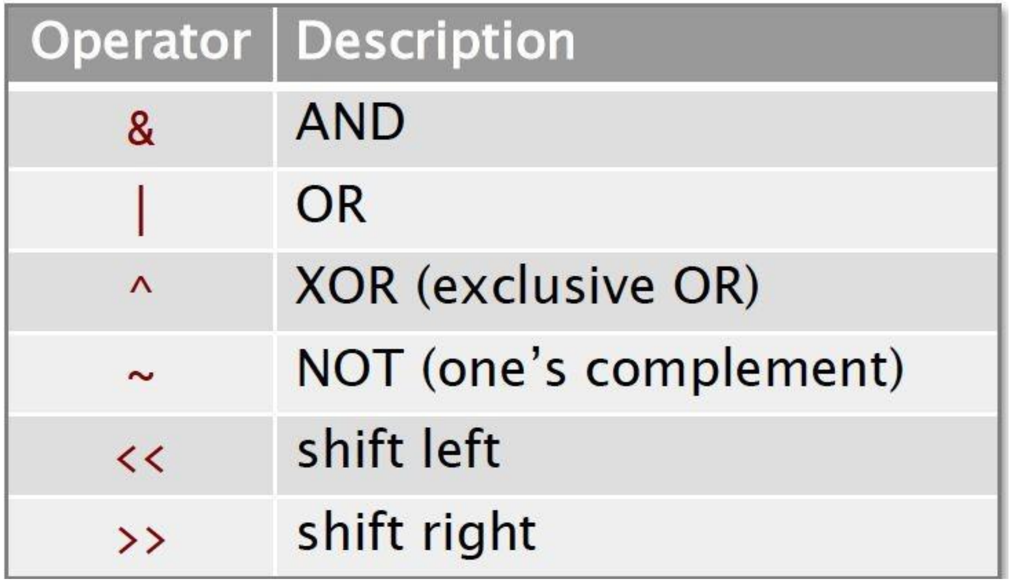
```
for (int8_t i = 7; i >= 0; i--) ...
```

- Use the appropriate literals
  - `1ULL` means 1 unsigned long long which is `uint64_t`
  - `1 << 63` can give unexpected results. A plain `1` is `int32_t`
- Never shift by more than the number of bits in the number
  - `((uint64_t) i) >> 64` is **undefined behavior (UB)**
- Never shift by a negative amount
  - Also **UB**

Two’s Complement (mentioned in lecture)

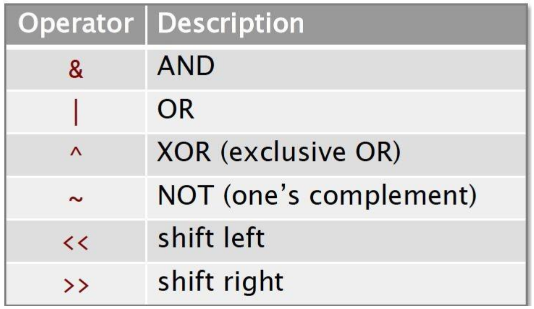
- Positive integers stay the same
- Negative integer: example to get -28
    - Start with positive integer 28: `00011100`
    - Flip the digit 0 to be 1 and vice versa: `11100011`
    - Add 1: `00000001`
    - We get -28: `11100100`
    - Check result:
        - `0 * 1 + 0 * 2 + 1 * 4 + 0 * 8 + 0 * 16 + 1 * 32 + 1 * 64 + 1 * -128 = -28`

Bit Hack Questions

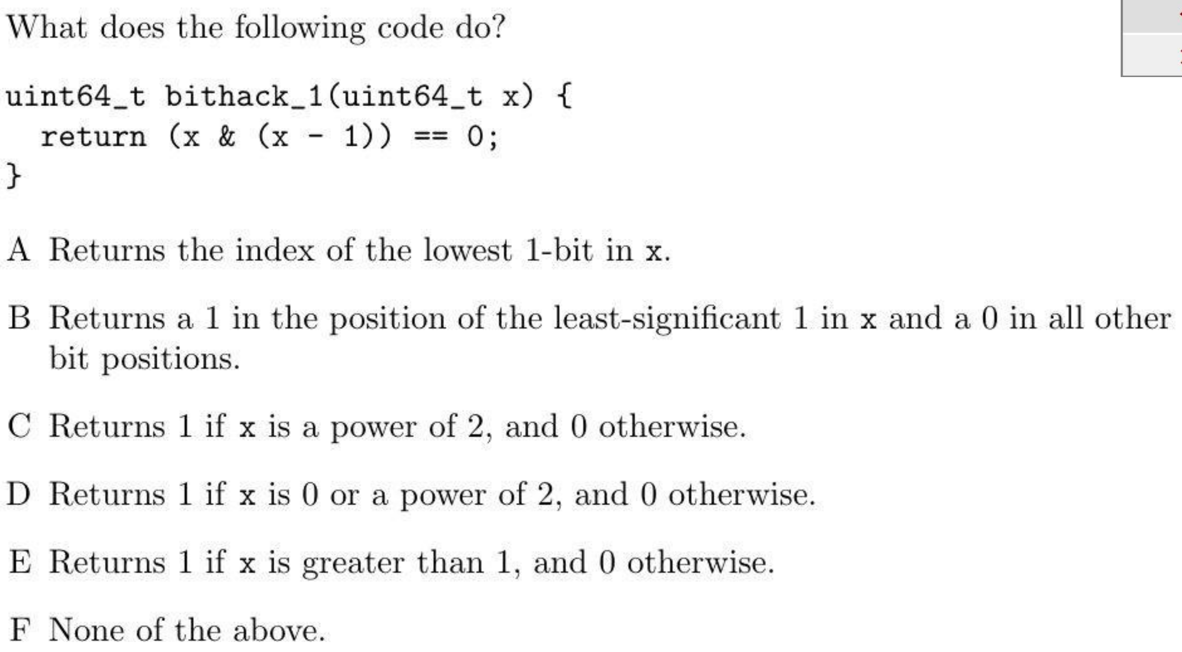


Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one’s complement)
<<	shift left
>>	shift right

Practice Question



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



What does the following code do?

```
uint64_t bithack_1(uint64_t x) {
  return (x & (x - 1)) == 0;
}
```

A Returns the index of the lowest 1-bit in `x`.

B Returns a 1 in the position of the least-significant 1 in `x` and a 0 in all other bit positions.

C Returns 1 if `x` is a power of 2, and 0 otherwise.

D Returns 1 if `x` is 0 or a power of 2, and 0 otherwise.

E Returns 1 if `x` is greater than 1, and 0 otherwise.

F None of the above.

Why?

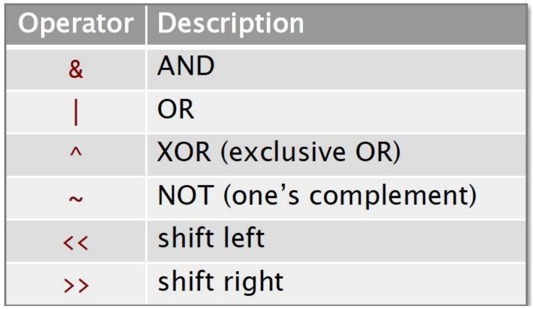
1. We know x -1 will find the first set bit on x, when scanning from the least significant bit and invert all bits till the found bit.
Ex: 1100**1000** – 1 = 1100**0111**
2. Now, let the number be x= abcd100…
3. x – 1 = abcd011….
4. x & x -1 = abcd000…
5. The above number will be 0 iff abcd are all 0.
6. Which means x must be of the form 00..010…
7. Which represents all and only powers of 2 or 0

Practice Question

This bit trick resembles the bit trick from lecture for computing $2^{\lceil \lg n \rceil}$, but all right shifts in that trick have been replaced with rightward bit rotations. For example, `0xDEADBEEF >> 12` yields `0x000DEADB`, and `rightrotate(0xDEADBEEF, 12)` produces `0xEEFDEADB`. For each of the assertions in the following the code, determine if the assertion would always succeed, if the assertion would always fail, or if the assertion would sometimes succeed and sometimes fail.

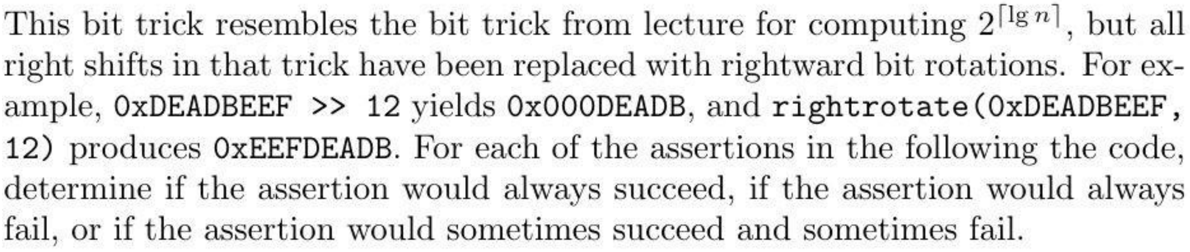
```
void bithack(uint64_t x) {
  uint64_t r = x - 1;
  r |= rightrotate(r, 1);
  r |= rightrotate(r, 2);
  r |= rightrotate(r, 4);
  r |= rightrotate(r, 8);
  r |= rightrotate(r, 16);
  r |= rightrotate(r, 32);
  r++;

  assert(r < 0);   // A.
  assert(r < 1);   // B.
  assert(r < 2);   // C.
  assert(r < 4);   // D.
}
```

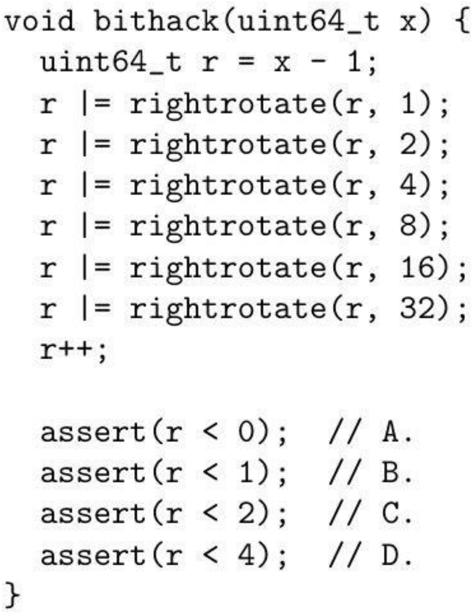


Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one’s complement)
<<	shift left
>>	shift right

Practice Question

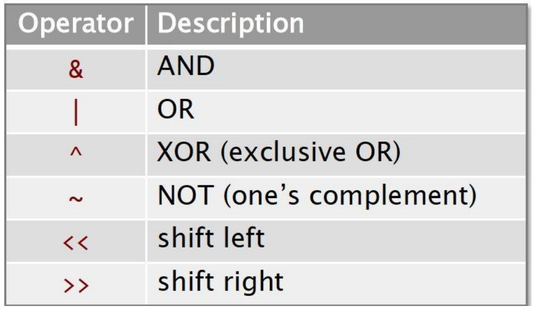


This bit trick resembles the bit trick from lecture for computing $2^{\lceil \lg n \rceil}$, but all right shifts in that trick have been replaced with rightward bit rotations. For example, `0xDEADBEEF >> 12` yields `0x000DEADB`, and `rightrotate(0xDEADBEEF, 12)` produces `0xEEFDEADB`. For each of the assertions in the following the code, determine if the assertion would always succeed, if the assertion would always fail, or if the assertion would sometimes succeed and sometimes fail.



```
void bithack(uint64_t x) {
  uint64_t r = x - 1;
  r |= rightrotate(r, 1);
  r |= rightrotate(r, 2);
  r |= rightrotate(r, 4);
  r |= rightrotate(r, 8);
  r |= rightrotate(r, 16);
  r |= rightrotate(r, 32);
  r++;

  assert(r < 0);   // A.
  assert(r < 1);   // B.
  assert(r < 2);   // C.
  assert(r < 4);   // D.
}
```

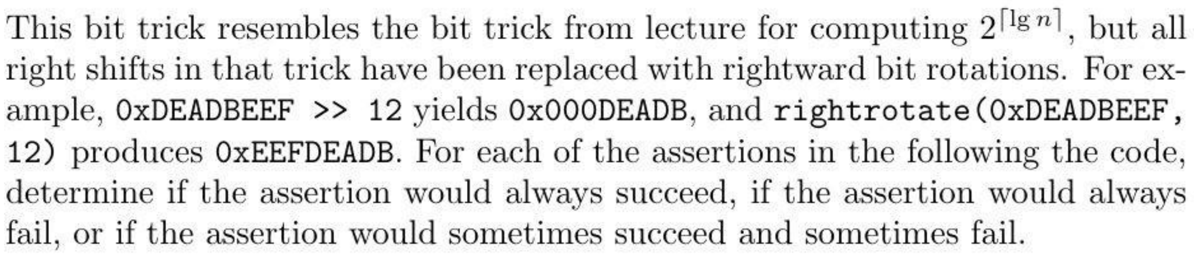


Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

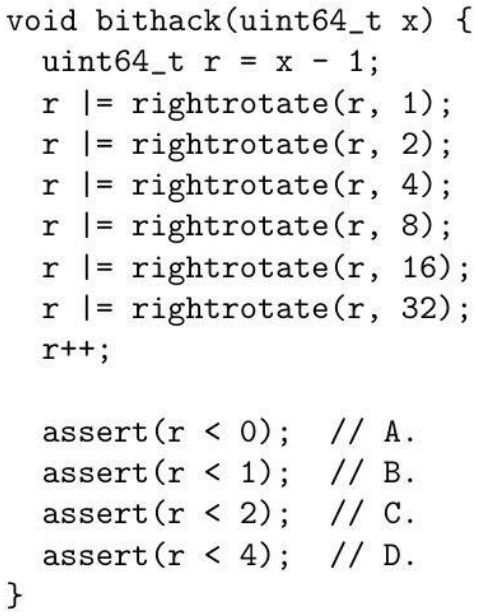
Trace: Take x = 2 => r =1
r = 100…01
r = 11100….01
r = 111111…001
.
.
.
r = 1111….1

r++ will result in 0
We can easily see that if any bit in x-1 is 1, then r is 0.
Otherwise, r is 1
So, it tests if x-1 is 0, that is if x is 1.

Practice Question



This bit trick resembles the bit trick from lecture for computing $2^{\lceil \lg n \rceil}$, but all right shifts in that trick have been replaced with rightward bit rotations. For example, `0xDEADBEEF >> 12` yields `0x000DEADB`, and `rightrotate(0xDEADBEEF, 12)` produces `0xEEFDEADB`. For each of the assertions in the following the code, determine if the assertion would always succeed, if the assertion would always fail, or if the assertion would sometimes succeed and sometimes fail.



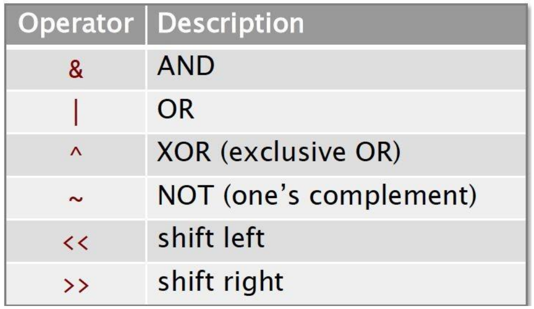
```
void bithack(uint64_t x) {
  uint64_t r = x - 1;
  r |= rightrotate(r, 1);
  r |= rightrotate(r, 2);
  r |= rightrotate(r, 4);
  r |= rightrotate(r, 8);
  r |= rightrotate(r, 16);
  r |= rightrotate(r, 32);
  r++;

  assert(r < 0);   // A.
  assert(r < 1);   // B.
  assert(r < 2);   // C.
  assert(r < 4);   // D.
}
```

Trace: Take x = 2 => r =1
r = 100…01
r = 11100….01
r = 111111…001
.
.
.
r = 1111….1

r++ will result in 0
We can easily see that if any bit in x-1 is 1, then r is 0. Otherwise, r is 1
So, it tests if x-1 is 0, that is if x is 1.

A – Never
B – Sometimes
C – True
D - True



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

Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

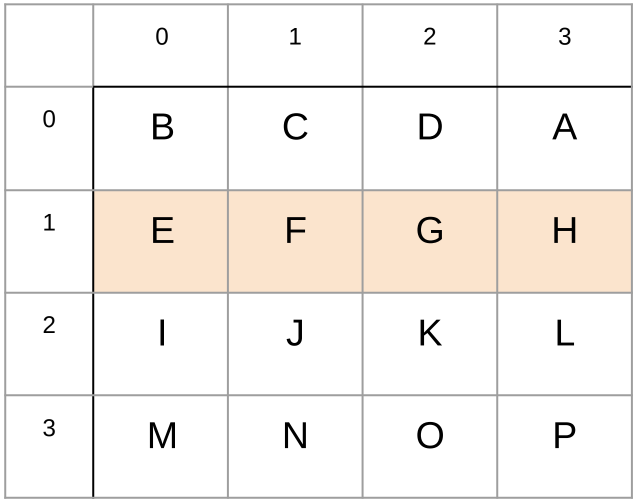
Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*

	0	1	2	3
0	B	C	D	A
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

Row-column-row

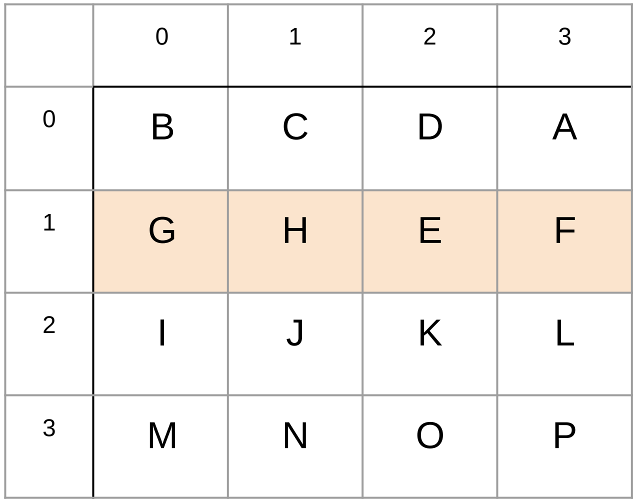
- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*



	0	1	2	3
0	B	C	D	A
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

Row-column-row

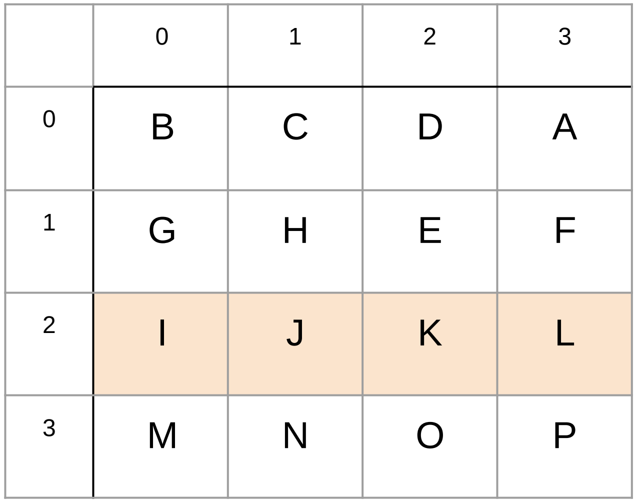
- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*



	0	1	2	3
0	B	C	D	A
1	G	H	E	F
2	I	J	K	L
3	M	N	O	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*



	0	1	2	3
0	B	C	D	A
1	G	H	E	F
2	I	J	K	L
3	M	N	O	P

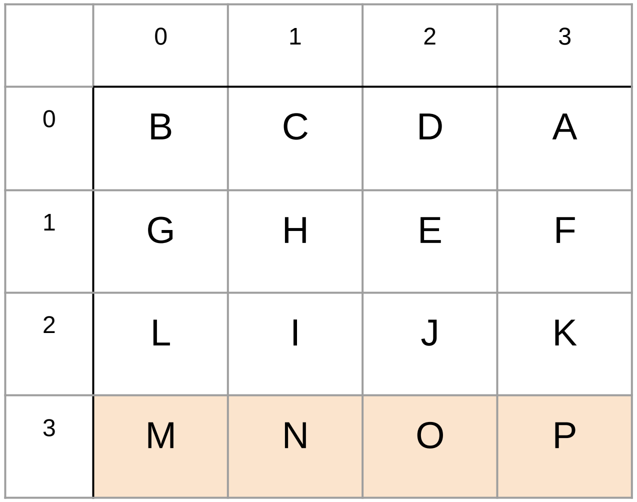
Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*

	0	1	2	3
0	B	C	D	A
1	G	H	E	F
2	L	I	J	K
3	M	N	O	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*



	0	1	2	3
0	B	C	D	A
1	G	H	E	F
2	L	I	J	K
3	M	N	O	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r* + *1*
    - Rotate col *c* down by *c* + *1*
    - Rotate row *r* left by *r*

	0	1	2	3
0	B	C	D	A
1	G	H	E	F
2	L	I	J	K
3	M	N	O	P

Rotate by 4 = no change

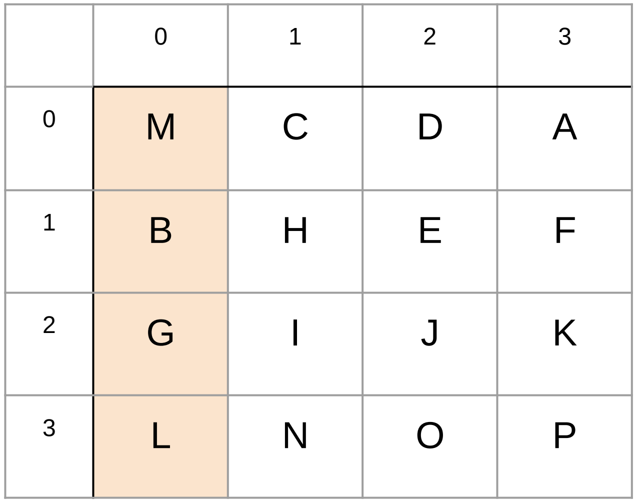
Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r* + *1*
    - Rotate col *c* down by *c* + *1*
    - Rotate row *r* left by *r*

	0	1	2	3
0	B	C	D	A
1	G	H	E	F
2	L	I	J	K
3	M	N	O	P

Row-column-row

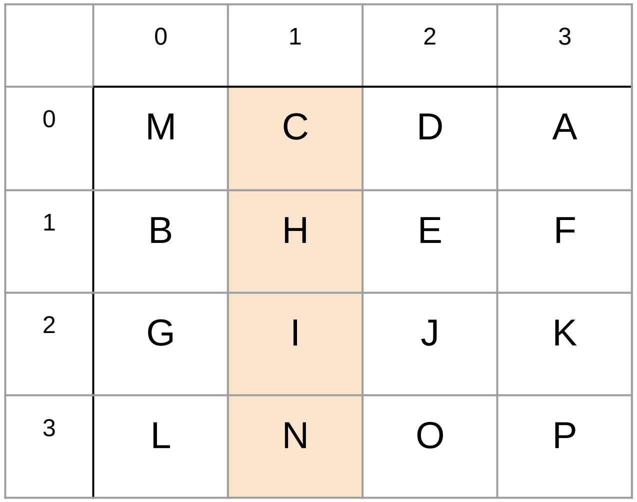
- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*



	0	1	2	3
0	M	C	D	A
1	B	H	E	F
2	G	I	J	K
3	L	N	O	P

Row-column-row

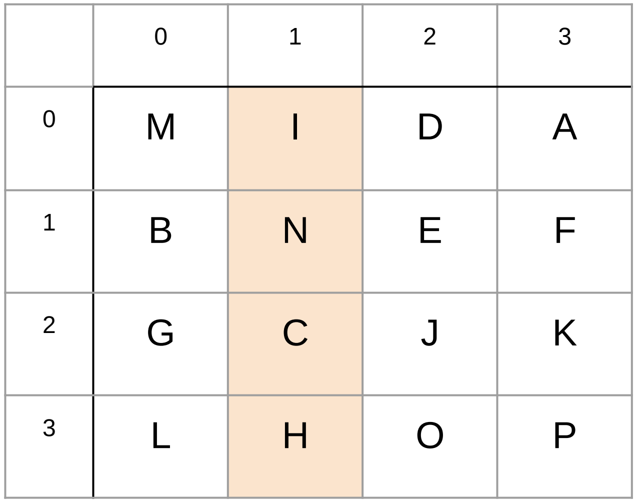
- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*



	0	1	2	3
0	M	C	D	A
1	B	H	E	F
2	G	I	J	K
3	L	N	O	P

Row-column-row

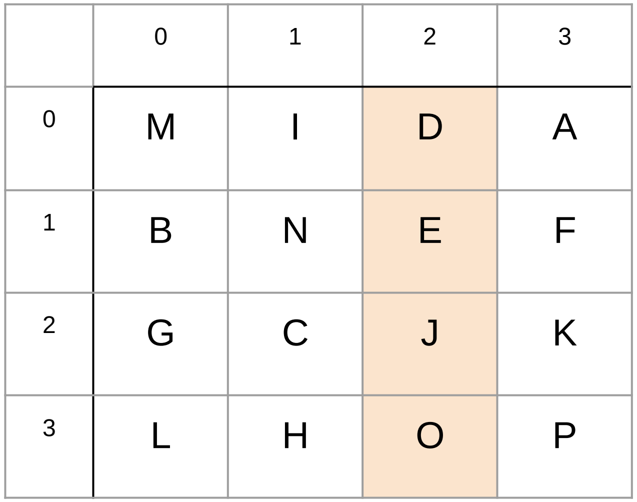
- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*



	0	1	2	3
0	M	I	D	A
1	B	N	E	F
2	G	C	J	K
3	L	H	O	P

Row-column-row

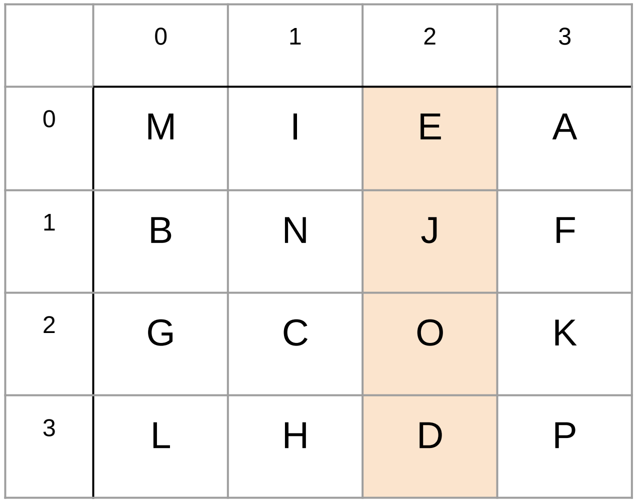
- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*



	0	1	2	3
0	M	I	D	A
1	B	N	E	F
2	G	C	J	K
3	L	H	O	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*



	0	1	2	3
0	M	I	E	A
1	B	N	J	F
2	G	C	O	K
3	L	H	D	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - ○ Rotate row *r* left by *r + 1*
    - ○ Rotate col *c* down by *c + 1*
    - ○ Rotate row *r* left by *r*

	0	1	2	3
0	M	I	E	A
1	B	N	J	F
2	G	C	O	K
3	L	H	D	P

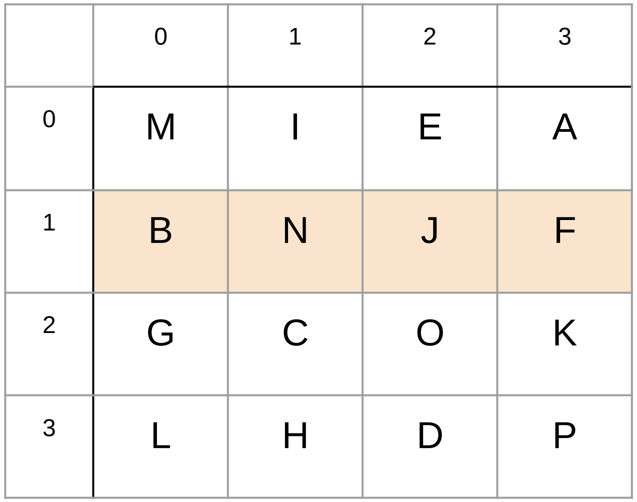
Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r* + *1*
  - Rotate col *c* down by *c* + *1*
  - Rotate row *r* left by *r*

	0	1	2	3
0	M	I	E	A
1	B	N	J	F
2	G	C	O	K
3	L	H	D	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row r left by $r + 1$
    - Rotate col c down by $c + 1$
    - Rotate row r left by r



	0	1	2	3
0	M	I	E	A
1	B	N	J	F
2	G	C	O	K
3	L	H	D	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*

	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	G	C	O	K
3	L	H	D	P

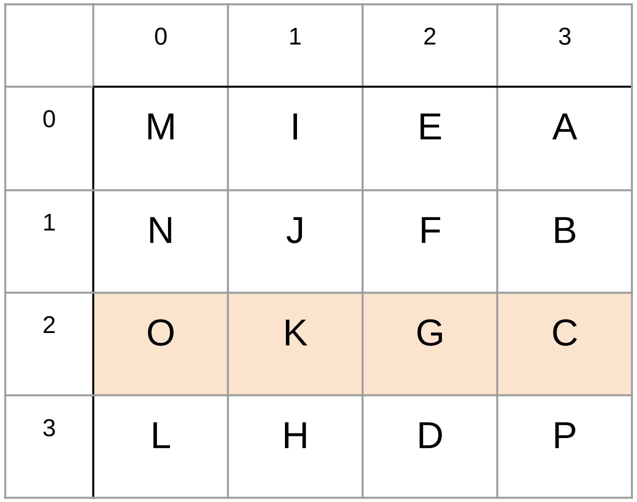
Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*

	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	G	C	O	K
3	L	H	D	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r* + *1*
    - Rotate col *c* down by *c* + *1*
    - Rotate row *r* left by *r*



	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	O	K	G	C
3	L	H	D	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
    - Rotate row *r* left by *r + 1*
    - Rotate col *c* down by *c + 1*
    - Rotate row *r* left by *r*

	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	O	K	G	C
3	L	H	D	P

Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*

	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	O	K	G	C
3	P	L	H	D

Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row *r* left by *r + 1*
  - Rotate col *c* down by *c + 1*
  - Rotate row *r* left by *r*

	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	O	K	G	C
3	P	L	H	D

Row-column-row

- Way of rotating block of bits
- Algorithm:
  - Rotate row r left by $r + 1$
  - Rotate col c down by $c + 1$
  - Rotate row r left by r

Original for Ref:

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	O	K	G	C
3	P	L	H	D

Row-column-row practice handout

- Practice executing the RCR algorithm on this handout.

ROTATING COLUMNS

Rotating Columns in Bit Matrix

Problem

Input: N × N matrix of bits, stored in row-major order.
Goal: Circularly rotate ith column of bits up i rows.

$$\begin{matrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{matrix} \quad \Rightarrow \quad \begin{matrix} a_{00} & a_{11} & a_{22} & a_{33} \\ a_{10} & a_{21} & a_{32} & a_{03} \\ a_{20} & a_{31} & a_{02} & a_{13} \\ a_{30} & a_{01} & a_{12} & a_{23} \end{matrix}$$

In the example that follows, we have N = 32. Each row is stored in a 32-bit word, with column 0 in the most-significant bit.

Naive Algorithm

```
const uint32_t N = 32;
const uint32_t mask = 1 << (N-1);
uint32_t A[N];
for (int i = 0; i < N; i++){
  uint32_t col = 0;
  // gather bits in column i
  for (int j = 0; j < N; j++)
    col = col | (((A[j] << i) & mask) >> j);
  // rotate bits in column i
  col = (col << i) | (col >> (N - i));
  // put column i back
  for (int j = 0; j < N; j++)
    A[j] = (A[j] & ~(mask >> i)) |
        (((col << j) >> i) & (mask >> i));
}
```

Naive Algorithm

```
const uint32_t N = 32;
const uint32_t mask = 1 << (N-1);
uint32_t A[N];
for (int i = 0; i < N; i++){
  uint32_t col = 0;
  // gather bits in column i
  for (int j = 0; j < N; j++)
    col = col | (((A[j] << i) & mask) >> j);
  // rotate bits in column i
  col = (col << i) | (col >> (N - i));
  // put column i back
  for (int j = 0; j < N; j++)
    A[j] = (A[j] & ~(mask >> i)) |
          (((col << j) >> i) & (mask >> i));
}
```

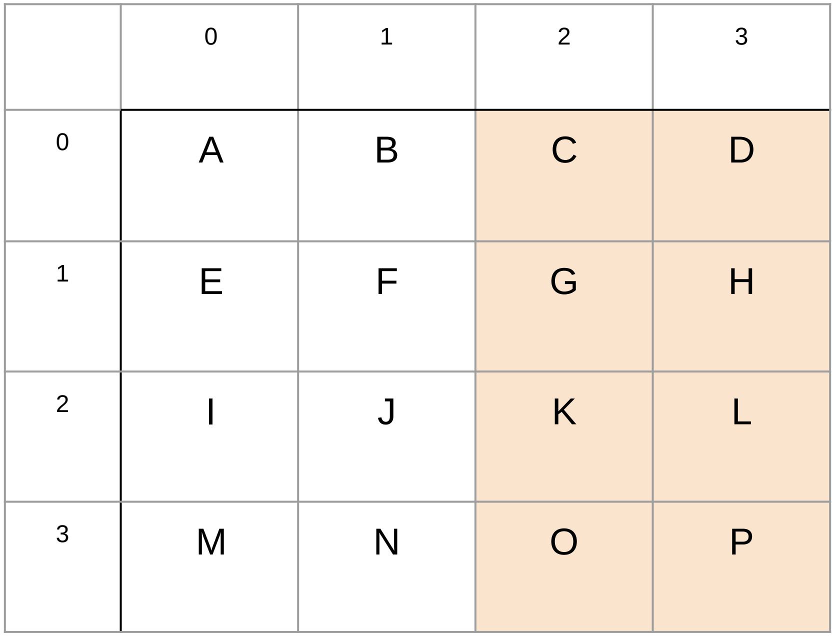
Work:
$\Theta(N^2)$.

Divide-and-Conquer Algorithm

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

Rotate columns 2 & 3 down by 2

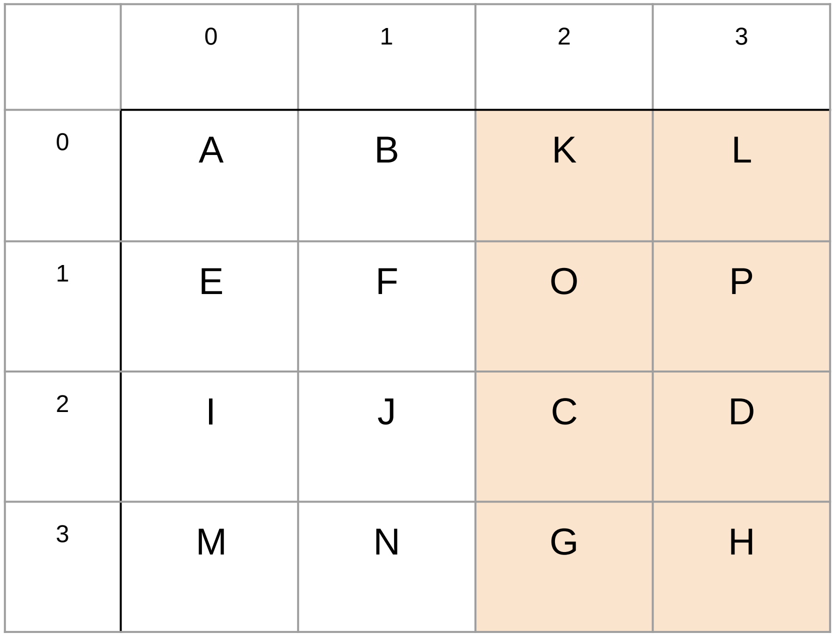
Divide-and-Conquer Algorithm



	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

Rotate columns 2 & 3 up by 2

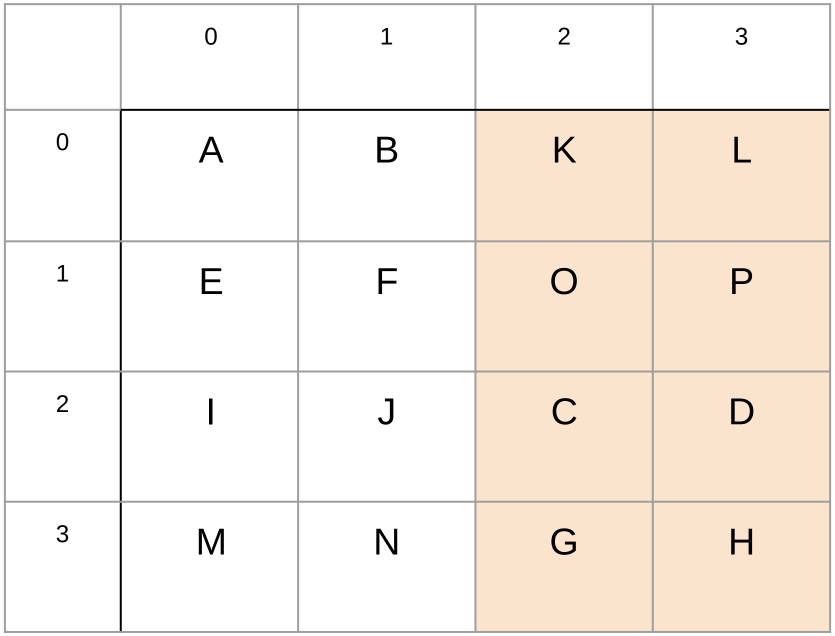
Divide-and-Conquer Algorithm



	0	1	2	3
0	A	B	K	L
1	E	F	O	P
2	I	J	C	D
3	M	N	G	H

Rotate columns 2 & 3 up by 2

Divide-and-Conquer Algorithm



	0	1	2	3
0	A	B	K	L
1	E	F	O	P
2	I	J	C	D
3	M	N	G	H

Rotate columns 2 & 3 up by 2

Divide-and-Conquer Algorithm

	0	1	2	3
0	A	B	K	L
1	E	F	O	P
2	I	J	C	D
3	M	N	G	H

Rotate columns 1 & 3 up by 1

Divide-and-Conquer Algorithm

	0	1	2	3
0	A	N	K	H
1	E	B	O	L
2	I	F	C	P
3	M	J	G	D

Rotate columns 1 & 3 up by 1

Divide-and-Conquer Algorithm

```c
uint32_t A[32], B[32]; // use B as scratch space

// rotate columns 16...31 up 16 positions
uint32_t stay_mask = 0xFFFF0000; // columns that don't move

for (int j = 0; j < 32; j++)
  B[j] = (A[j] & stay_mask) | (A[(j+16) % 32] & ~stay_mask);

// rotate columns 8..15 and 24...31 up 8 positions
stay_mask = 0xFF00FF00;

for (int j = 0; j < 32; j++)
  A[j] = (B[j] & stay_mask) | (B[(j+8) % 32] & ~stay_mask);
```

Divide-and-Conquer Algorithm

```
uint32_t A[32], B[32]; // use B as scratch space

// rotate columns 16...31 up 16 positions
uint32_t stay_mask = 0xFFFF0000; // columns that don't move

for (int j = 0; j < 32; j++)
  B[j] = (A[j] & stay_mask) | (A[(j+16) % 32] & ~stay_mask);

// rotate columns 8..15 and 24...31 up 8 positions
stay_mask = 0xFF00FF00;

for (int j = 0; j < 32; j++)
  A[j] = (B[j] & stay_mask) | (B[(j+8) % 32] & ~stay_mask);
```

Work: $\Theta(N \lg N)$

Divide-and-Conquer Algorithm

```
uint32_t A[32], B[32]; // use B as scratch space

// rotate columns 16...31 up 16 positions
uint32_t stay_mask = 0xFFFF0000; // columns that don't move

for (int j = 0; j < 32; j++)
  B[j] = (A[j] & stay_mask) | (A[(j+16) % 32] & ~stay_mask);

// rotate columns 8..15 and 24...31 up 8 positions
stay_mask = 0xFF00FF00;

for (int j = 0; j < 32; j++)
  A[j] = (B[j] & stay_mask) | (B[(j+8) % 32] & ~stay_mask);
```

Work: $\Theta(N \lg N)$

But sometimes the asymptotically best algorithms don’t perform well in practice. You must decide for yourself.