One-Week Course: Mastering Bit Twiddling Hacks

Course Overview: This 7-day course covers the famous collection of bit manipulation tricks known as **Sean Eron Anderson's "Bit Twiddling Hacks"**. Each day we focus on a coherent subset of these hacks, building from fundamental concepts to advanced techniques. The course is designed for intermediate programmers familiar with binary numbers, bitwise operators, and basic C/C++.

Throughout the week, we'll explore how these hacks work, why they work, and how to implement them in code (primarily C/C++, with occasional Python analogues). We include clear explanations, step-by-step breakdowns, annotated examples, real-world use cases, and practice problems. You'll also get tips for visualizing bit operations, debugging, and avoiding common pitfalls. By week's end, you'll have a solid grasp of clever bit-level techniques to write faster and branch-free code.

Table of Contents

Day 1: Bit Sign and Basic Property Tricks

Topics: Determining the sign of an integer, checking opposite signs, sign extension, and checking if a number is a power of 2. Today introduces fundamental bit hacks that query properties of numbers without branching.

1.1 Computing the Sign of an Integer (–, 0, or +)

What it does: Determines whether an integer `v` is negative, zero, or positive, producing an output sign value (e.g. -1 for negative, 0 for zero, +1 for positive).

How it works: Instead of using an `if` to check the sign, we can leverage how negative numbers are represented in two's complement. In C, the expression `(v < 0)` evaluates to `1` if `v` is negative and `0` otherwise. By negating this, we get -1 for negative inputs and 0 for non-negative. Another trick uses bit shifting: a right arithmetic shift by 31 (for 32-bit ints) will fill with the sign bit (1 for negatives, 0 for non-negatives), resulting directly in -1 or 0.

Example – C Implementation:

c

```c
int v = /* some integer */;
int sign;

// Method 1: Portable branchless sign (gives -1 for negative, 0 for >=0)
sign = -(v < 0);

// Method 2: Using bit-shift (faster but non-portable if right shift is not arithmetic
sign = v >> (sizeof(int)*CHAR_BIT - 1);

printf("Sign of %d is %d\n", v, sign);
```

This code sets `sign` to -1 if `v` is negative, otherwise 0. The shift method relies on an *arithmetic* right shift that replicates the leftmost bit (the sign bit) across the shifted bits; in two's complement, -1 is represented as all 1s in binary, so a negative `v` shifted right yields -1. (Caution: The C standard leaves arithmetic shifts of negative values implementation-defined, so method 1 is safer for portability.)

If we want the sign in {-1, 0, +1}, a neat idiom is:

c

```c
sign = (v > 0) - (v < 0);
```

This yields +1 for positive, -1 for negative, and 0 for zero. It works because `(v > 0)` is 1 for positive (0 otherwise) and `(v < 0)` is 1 for negative (0 otherwise), so the subtraction gives 1-0=1 for positive, 0-1=-1 for negative, and 0-0=0 for zero.

Use Cases: Quickly obtaining the sign without branching is useful in performance-critical code (e.g. vectorized math, games or physics engines) to avoid pipeline stalls from branch misprediction. It's also useful for implementing functions like `sgn()` or normalizing values to -1,0,1.

Practice Exercises:

- Implement the sign function in C using the above methods and verify it for various inputs (including edge cases like `INT_MIN` and `INT_MAX`).
- In Python, mimic this behavior (Python doesn't have fixed 32-bit ints, but you can simulate this using conditional expressions or by forcing a bit-width).
- Modify the code to produce +1 for non-negative and 0 for negative (hint: use XOR as shown on the Bit Twiddling Hacks page).

1.2 Detecting Opposite Signs of Two Integers

What it does: Checks if two integers `x` and `y` have opposite signs (one positive, one negative).

How it works: The bit trick is based on the sign bit (the MSB in two's complement). If `x` and `y` have opposite signs, then one of them has the MSB 1 and the other 0. XORing them (`x ^ y`) will have the MSB = 1 in this case (because 1 xor 0 = 1) and = 0 if they have the same sign (1 xor 1 = 0, or 0 xor 0 = 0). Therefore, `(x ^ y) < 0` will be true iff the sign bits differ.

Example – C Implementation:

```c
int x = /*...*/, y = /*...*/;
bool opposite = ((x ^ y) < 0);
printf("%d and %d have opposite signs? %s\n", x, y, opposite ? "yes" : "no");
```

This sets `opposite` to true if one of the integers is negative and the other is non-negative. Internally, `x ^ y` performs a bitwise XOR; if the result is negative, it means the resulting MSB is 1, which only happens when the inputs' MSBs differed.

Use Cases: A quick check for opposite sign is handy in algorithms that need to detect sign changes (e.g. root-finding, slope changes) without branching. It's also used in overflow detection for addition (if adding two numbers of opposite signs, overflow can't happen in two's complement).

Practice Exercises:

- Write a function `bool opposite_signs(int x, int y)` using the XOR trick. Test it with pairs of integers (try positive/negative pairs, and also include zero).
- Explain why this trick means zero is considered "positive" in your logic (hint: 0 and a negative number will appear as opposite by the XOR test because 0 has MSB 0 like a positive). How might you modify the check to treat 0 as neutral or not trigger a false "opposite" in certain contexts?

1.3 Sign Extension to a Wider Type

What it does: Sign extension takes a number represented in a smaller bit-width and extends it to a larger width (e.g., 8-bit to 32-bit) while preserving its sign (so that a negative small value remains negative when widened).

How it works: Suppose we have an `b`-bit signed value `x` stored in an `int` (with garbage or zero in higher bits). We want to extend the sign bit at position b-1 (0-indexed) to the full `int`. A trick for sign extension is to create a mask from the bit at position b-1, then use it to flip and subtract. For example:

1. Compute a mask `m = 1U << (b - 1)`. This mask has a 1 at the original sign bit position (bit *b-1*) of `x`.

2. XOR the value with the mask: `x ^ m`. If `x`'s sign bit was 1 (meaning original number was negative in b-bit), this will flip that bit to 0; if it was 0 (number was non-negative), it flips it to 1.
3. Subtract the mask: `(x ^ m) - m`. If original sign was 1, then after XOR the high bit is 0 and subtracting `m` will borrow, effectively turning all higher bits into 1s (yielding a negative extended value). If original sign was 0, XOR made the bit 1 but subtracting brings it back to 0 without borrow, leaving higher bits 0.

The result is the properly sign-extended integer.

Example – C Implementation:

```c
unsigned b = 5;        // say x is 5-bit signed (range -16..15)
int x = 0x0E;          // 5-bit pattern 0x0E (01110b) = 14 in decimal, stored in int
int m = 1U << (b - 1); // mask for bit 4 (0-indexed)
int r = (x ^ m) - m;   // sign-extend 5-bit value to int
printf("Extended value = %d\n", r);
```

For the example above, `x = 0x0E` (binary `01110`) has sign bit 0 (it's positive), so `m = 0x10` (binary `10000`). `x ^ m` = `11110` and `(x^m) - m` = `11110 - 10000` = `01110` (which is 14, unchanged as expected). If instead `x = 0x12` (binary `10010`, which in 5-bit should represent -14 since the sign bit is 1), then `m = 10000`, `x ^ m = 00010`, and `(x^m) - m = 00010 - 10000 = 10010` in the 32-bit result (which is -14 in decimal, as desired).

Another method is to left-shift then right-shift the value by the remaining bits. For instance, if `int` is 32-bit and `b=5`, we can do:

```c
int r2 = (x << (32 - b)) >> (32 - b);
```

This shifts the 5-bit number to the top of the 32-bit `int`, then right shifts back, relying on the arithmetic right shift to propagate the sign bit. This is efficient (just 2 operations) but non-portable in strict C (again due to implementation-defined behavior of shifting negatives). The XOR/subtract method works purely with unsigned operations for the mask, making it more portable.

Use Cases: Sign extension is common when dealing with packed data or lower-precision values. For example, extracting a 5-bit field from a 32-bit register and interpreting it as a signed value, or reading 8-bit/16-bit values and needing them in 32-bit form. Bit hacks for sign extension avoid branching and make this conversion very fast.

Practice Exercises:

- Write a function `int sign_extend(int x, unsigned b)` that sign-extends the lower `b` bits of `x` to a full 32-bit int using the XOR/subtract trick. Test it with various values (especially the case where sign bit =1 and all other lower bits =0, which is the most negative number in that range).
- Verify that for all values in the range of `b` bits, the result matches what you get by manually converting the binary to a signed number. (Tip: for debugging, print the binary representations or use Python's bit_length and bit operations to simulate.)
- (Advanced) Research what happens if you try to sign-extend using bit-fields in C or using arithmetic shifts, and understand the potential portability issues. How does languages like Java (which specify arithmetic shifts) or Python (infinite precision ints) differ here?

1.4 Determining if an Integer is a Power of 2

What it does: Checks if an unsigned integer `v` is a power of two (e.g. 1, 2, 4, 8, …) **and non-zero**. Powers of 2 in binary have exactly one bit set (e.g. 1 = `0001`, 2 = `0010`, 4 = `0100`, etc.).

How it works: A classic bit trick uses the property above: **a number `v` is a power of two if and only if `v` has one bit set**. An elegant check is:

```
v > 0 && (v & (v - 1)) == 0
```

When you subtract 1 from a power of 2, you get a number with all 1s in the positions lower than that single set bit. For example, `4 (100)_2 - 1 = 3 (011)_2`. ANDing the original with one less:

- If `v` was power of 2: `v` in binary = `100...0`, and `v-1 = 011...1`. Their AND is 0.
- If `v` was not a power of 2 (had more than one 1 bit), subtracting 1 will not clear all bits; some 1s remain in common positions, so AND is non-zero.

Example – C Implementation:

```c
unsigned int v = /*...*/;
bool isPowerOf2 = (v != 0) && !(v & (v - 1));
printf("%u is power of 2? %s\n", v, isPowerOf2 ? "yes" : "no");
```

This sets `isPowerOf2` true if `v` is a power of 2. We check `v != 0` to exclude 0 (since 0 & (0-1) also yields 0, but 0 is not a power of 2 by definition). The expression `(v & (v-1))` will be 0 for powers of 2, non-zero otherwise.

Use Cases: Determining power-of-two-ness is useful in algorithms that optimize for powers of 2 (e.g. certain FFT lengths, memory alignments, hashing, etc.), or when using bit masks and need to ensure only one flag is set.

Common Pitfall: Make sure to exclude 0, since `0 & -1 = 0` would incorrectly indicate "true". Also note this trick typically assumes `v` is unsigned; if `v` is signed and negative, the expression makes little sense (you may want to cast to unsigned or handle negatives separately if needed).

Practice Exercises:

- Implement `bool is_power_of_two(unsigned v)` using the above trick and test it on a range of values (0, 1, 2, 3, 4, 16, 18, etc.).
- Using this property, write a loop to print all powers of 2 within a 32-bit range.
- (Thought experiment) Why does this trick work mathematically? Write out a few examples in binary to see the pattern (e.g. for v=8, v-1=7, and for v=10 which is not a power of 2, see what v & (v-1) yields).

Day 1 Summary: We explored basic bit hacks that check number properties using bitwise logic. We saw how to get the sign of a number, detect opposite signs, extend signed values to larger sizes, and test for power-of-two values – all without branches. You should practice reading binary representations and tracing through these operations, as understanding the pattern of bits is key to demystifying bit tricks. In the next lesson, we will apply similar thinking to perform arithmetic and conditional operations with bits instead of branches.

Day 2: Branchless Arithmetic and Conditional Operations

Topics: Computing absolute value and min/max without branching, using bits to conditionally set/clear/negate values, and merging bits from two values according to a mask. Today's focus is eliminating conditional jumps (`if/else`) by using clever bit manipulations.

2.1 Absolute Value of an Integer (without branching)

What it does: Computes the absolute value `|v|` of an integer without using conditional logic. Normally one might write `if (v < 0) v = -v;`, but that branch can be avoided.

How it works: The trick is to create a mask that is all 1s (i.e. `0xFFFFFFFF`...) if `v` is negative, or all 0s if `v` is non-negative. Then we can use this mask to select either `v` or `-v`. Here's one method:

1. Compute `mask = v >> (sizeof(int)*CHAR_BIT - 1)`. As discussed on Day 1, this is -1 (all bits 1) if `v` is negative, or 0 if `v` is non-negative.
2. Using that mask, we can get absolute value as: `r = (v + mask) ^ mask`.

Let's break down `r = (v + mask) ^ mask` for the two cases:

- If `v` is negative: then `mask = -1`. So `v + mask` is `v - 1`. XORing that with `mask` (i.e. XOR with ~0) flips all bits of `(v-1)`. In two's complement, `~(v-1)` equals `-v` (this is a known identity: `-v

`= ~v + 1`), so `~(v-1) = ~v + ~(-1) = ~v + 0 = ~v`, wait, let's do it carefully: `r = (v-1) ^ 0xFFFFFFFF`. Flipping all bits of `v-1` is the same as `~(v-1)`. Since `-v = ~(v) + 1`, it can be shown that `~(v-1) = -v` for negative v). Thus we get `-v`, the absolute value.

- If `v` is non-negative: `mask = 0`. Then `v + mask = v`, and `v ^ mask = v ^ 0 = v`. So it leaves `v` unchanged, which is correct for absolute value.

Another way to see it: `(v + mask) ^ mask` effectively does `v` XOR `mask` then plus `mask` (one can also write it as `(v ^ mask) - mask` which is a common variation). The mask acts like a conditional negator: when mask is all 1s, it flips bits and subtracts 1 (which is two's complement negation), and when mask is 0, it does nothing.

Example – C Implementation:

```c
int v = /* some int */;
int mask = v >> (sizeof(int)*CHAR_BIT - 1);
int abs_v = (v + mask) ^ mask;
// (Alternatively: int abs_v2 = (v ^ mask) - mask; does the same)
printf("|%d| = %d\n", v, abs_v);
```

For example, if `v = -5`: in 32-bit, `v = 0xFFFFFFFB`, so `mask = 0xFFFFFFFF`. Then `v + mask = 0xFFFFFFFA` and XOR with `mask` gives `0x00000005` which is 5. If `v = 5`: `mask = 0x0`, `v+mask = 5`, XOR with 0 stays 5.

Use Cases: Calculating absolute values frequently (e.g. in DSP or graphics code) where branching could hurt performance. This trick may also be used in vectorized instructions or SIMD, where you can apply a mask to many values at once.

Pitfalls:

- The expression will yield undefined behavior on the minimum representable integer (e.g. `INT_MIN` = -2147483648 for 32-bit) because the positive value 2147483648 cannot be represented in a 32-bit signed int. However, this is a limitation of absolute value itself (even a normal `if` can't make -INT_MIN positive in a 32-bit int). In practice, the above code will give `INT_MIN` back for `INT_MIN` input, which is something to be aware of if it matters to your application.
- Remember that the mask creation uses an arithmetic shift. On non-two's-complement systems or where `>>` isn't arithmetic for signed types, you'd need to adjust (the page suggests a fully portable approach using casting to unsigned for shifting, but in typical two's complement systems this is fine).

Practice Exercises:

- Implement the branchless absolute value and test it against `abs()` from `<stdlib.h>` for all integers in a certain range (you can loop from -10000 to 10000 to be confident). Check if results differ for `INT_MIN`.
- Try to derive the formula `(v + mask) ^ mask` yourself starting from the idea of using -1/0 mask. Why do we add before XOR? Could you XOR first then add? (Hint: try `(v ^ mask) - mask` which is an equivalent form and see if it makes sense).
- In Python, since integers are unbounded, this specific trick isn't needed (Python's `abs()` will handle big ints), but simulate it by forcing a fixed bit-width (e.g., mask with `0xFFFFFFFF` to simulate 32-bit overflow) to see the effect.

2.2 Minimum or Maximum of Two Integers (without branching)

What it does: Computes the minimum or maximum of two values `x` and `y` without using comparison branches.

How it works: A known idiom is:

```c
min = y ^ ((x ^ y) & -((x < y) ? 1 : 0));
max = x ^ ((x ^ y) & -((x < y) ? 1 : 0));
```

This looks complex, but let's unpack the `min` formula `r = y ^ ((x ^ y) & -(x < y))`:

- `(x < y)` evaluates to 1 if `x < y` else 0.
- A Boolean `true/false` in C can be treated as 1 or 0. We negate it (with unary `-`), so we get all bits =1 (i.e. -1) if `x < y`, or 0 if `x >= y`.
- `(x ^ y)` gives a bit mask of the bits that differ between `x` and `y`.
- If we AND that with either 0 or -1 (depending on `x<y`), it will either zero out all differing bits or keep them.
- Finally XOR with `y` will selectively choose bits from either `x` or `y`.

In effect, when `x < y` (meaning `x` should be the min), `-(x<y)` = -1, so `(x ^ y) & -1` = `(x ^ y)`. Then `y ^ (x ^ y)` = `x` (because `a ^ b ^ a = b` – here `b=x^y` and `a=y`). So `r = x` as expected for min. When `x >= y`, `-(x<y)` = 0, so the expression becomes `y ^ 0 = y`. Thus `r = y` (which is the smaller if x>=y). The formula for max simply swaps roles (or you could do `max = x ^ y ^ min` once you have min).

Example – C Implementation (Min):

c

```c
int x = /*...*/, y = /*...*/;
int min = y ^ ((x ^ y) & -((x < y) ? 1 : 0));
int max = x ^ ((x ^ y) & -((x < y) ? 1 : 0));
// Alternatively, once min is computed: max = x ^ y ^ min;
printf("min=%d, max=%d\n", min, max);
```

Let's test a scenario: `x=5, y=9`. Here `x<y` is true (1).

- `x ^ y` = 5 ^ 9 = 12 (binary 0101 ^ 1001 = 1100).
- `-(x<y)` = -1 (all 1s).
- `(x^y) & -1` = 12 (no change).
- `y ^ 12` = 9 ^ 12 = 5, which is min as expected. For max, the same expression gives 9.

If `x=10, y=3` (`x<y` false):

- `x^y` = 9 (1010 ^ 0011 = 1001).
- `-(x<y)` = 0.
- Masked `&0` gives 0.
- `y ^ 0` = 3, which is min (since y is smaller here). Max would yield 10.

Another implementation uses a different approach with arithmetic shifts:

c

```c
int diff = x - y;
min = y + ((diff) & ((diff) >> (sizeof(int)*CHAR_BIT - 1)));
max = x - ((diff) & ((diff) >> (sizeof(int)*CHAR_BIT - 1)));
```

In this form, `(diff >> 31)` produces -1 if `x < y` (because then diff is negative) or 0 if `x >= y`. That mask `& diff` either subtracts nothing or subtracts the difference to get the min. This avoids the conditional operator but relies on arithmetic shift behavior.

Use Cases: Branchless min/max is useful in performance-sensitive code or to avoid side-channel variations (in cryptography, constant-time code often avoids branches). It can also be used in vectorized computations.

Pitfalls:

- As with sign, the shift method relies on defined behavior of right shifting negative numbers (which is fine on two's complement systems).

- Also, ensure the difference `x - y` does not overflow (if using unsigned or carefully considering limits, it's usually fine for signed 32-bit as overflow is defined for unsigned only; for signed, overflow is undefined, but in this context, if x and y are within normal range, diff fits in signed range except possibly extreme cases).
- Code clarity: these formulas are quite inscrutable at first glance. Always comment such code in real projects!

Practice Exercises:

- Implement a function `int min_no_branch(int x, int y)` using the XOR/AND method. Test it exhaustively for a range of values (you can even brute-force all `x,y` in a small range, or use random testing against `std::min` to be sure). Do the same for max.
- Try to explain in your own words or with a diagram why the XOR method works. Specifically, trace it with binary examples as above.
- Compare the performance of your branchless min with a regular `return x < y ? x : y;` by timing a large loop of each. You might find that modern CPUs are so good at branch prediction that the difference is negligible or even that the branching version can be just as fast or faster unless branch misprediction happens.

2.3 Conditionally Setting or Clearing Bits

What it does: Sets or clears certain bits in a word depending on a condition, without branching. For example: *"if flag `f` is true, set bits in mask `m` in word `w`; if `f` is false, clear those bits."*

How it works: A straightforward way is: `w = f ? (w | m) : (w & ~m)`. The bit-hack way uses a combination of bitwise ops to avoid the `?:`. One slick formula is:

```
w ^= (-f ^ w) & m;
```

Assume `f` is 1 for true, 0 for false (e.g., a `bool` or int that is 0/1). Let's analyze `w ^= (-f ^ w) & m`:

- If `f` is 1 (true): `-f` = -1 (all bits 1). Then `(-f ^ w)` = `(~0 ^ w)` = `~w`. So we have `w ^= (~w & m)`. `(~w & m)` equals `m` minus any bits that are already 1 in `w` (it picks bits from `m` that *need* setting). So `w ^= ...` will flip those bits to 1 (since they were 0). In effect, this sets the bits in `m` (and leaves others).
- If `f` is 0 (false): `-f` = 0. Then expression becomes `w ^= (0 ^ w) & m` = `w ^= (w & m)`. That will toggle off (clear) any bits that are 1 in both `w` and `m` (because `w & m` picks out the bits that are set in `w` that we might want to clear, and XORing those with `w` will turn them to 0). Bits not in `m` or not set in `w` remain unchanged. So it clears the mask bits.

Thus, the single formula can both set or clear bits based on `f`. Another equivalent form is:

```
w = (w & ~m) | (-f & m);
```

This might be easier to understand: if `f=1`, `-f & m` = `m` (so we OR in the mask), and `w & ~m` zeroes out those bits first. If `f=0`, `-f & m` = 0 (so OR adds nothing), and `w & ~m` clears the bits. This form may produce slightly more operations on some CPUs, but conceptually similar.

Example – C Implementation:

```c
unsigned int w = 0xACE0;    // some pattern
unsigned int m = 0x00F0;    // mask of bits to set/clear (here bits 4-7)
bool f = /* condition flag */;

w ^= ((-f) ^ w) & m;  // set bits [4-7] if f=1, clear them if f=0

// Check result:
printf("Result = 0x%X\n", w);
```

If initially `w = 0xACE0` (1010 1100 1110 0000 in binary) and `m = 0x00F0` (0000 0000 1111 0000), and say `f=true` (1), then `(-f ^ w) & m = (~w) & m`. `~w = 0101 0011 0001 1111`. Masking that with `m` yields `0000 0000 0001 0000` (the bit 4 that was 0 in w). XORing that into w sets bit 4. If `f=false` (0), then `(-f ^ w) & m = (w & m)`, which would be `0000 0000 1110 0000` and XORing that into w would turn off bits 5-7 (since they were 1 in w).

Use Cases: Toggling specific bits on or off is common in low-level programming (device registers, flags, permission bits, etc.). Doing it branchlessly can be important in concurrent programming or signal handling where you want atomic-like operations or simply to avoid jumps.

Practice Exercises:

- Given a byte `w`, write a function to set or clear its high-order bit (0x80) based on a flag, using the above method. Test for both outcomes.
- Try to derive the formula `w ^= (-f ^ w) & m` from the truth table of `f`. Consider splitting it into two cases and then finding a single expression that covers both.
- The alternate form `(w & ~m) | (-f & m)` might be more readable – implement and verify it yields the same results. Count the number of operations in each form to see which might be more efficient.

2.4 Conditionally Negating a Value

What it does: Negates a value only if a certain flag is true (or false), without branching. For example, compute `r = f ? -v : v` in a branch-free manner.

How it works: This is similar to above, but now we want to either flip all bits and add 1 (negate) or do nothing. We can use a mask derived from the flag. Two patterns from the Bit Hacks collection are:

- If we want to negate *only when* a flag is **false** (`fDontNegate` means "don't negate if true"):

  c

  ```c
  r = (fDontNegate ^ (fDontNegate - 1)) * v;
  ```

  This one is a bit arcane. Let's decipher: If `fDontNegate` is true (1), then `fDontNegate - 1 = 0`, so expression becomes `(1 ^ 0) * v = 1 * v = v` (no negation). If `fDontNegate` is false (0), then `0 ^ (-1) = -1` (all bits 1), so we get `-1 * v = -v`. Thus it conditionally negates when the flag is false. The multiplication by -1 is effectively negation.

- If we want to negate when a flag is **true** (`fNegate` indicates we should negate):

  c

  ```c
  r = (v ^ -fNegate) + fNegate;
  ```

  Here, if `fNegate=1`: `-fNegate = -1` (all bits 1). So `v ^ -1 = ~v`, then adding `fNegate` (which is 1) yields `~v + 1`, which is -v (two's complement negation). If `fNegate=0`: `-fNegate = 0`, so `v ^ 0 = v`, plus 0 stays `v`. So it negates only when flag is true.

Example – C Implementation:

c

```c
int v = 10;
bool fNegate = true;  // if true, we want to negate v

int r = (v ^ -fNegate) + fNegate;
printf("Conditional negation result: %d\n", r);
```

In this example, `fNegate=true`, so it should compute `-10` which is printed as -10. If `fNegate=false`, it would leave `v` as is (10). You can test the other formula as well by setting `fDontNegate = !fNegate` appropriately.

Use Cases: This is useful when you have to apply a sign or factor conditionally. For instance, say you have a value and you need to negate it based on some condition (like an axis direction in graphics, or an adjustment in an algorithm) but want to avoid branching – these formulas accomplish that.

Practice Exercises:

- Using the second formula, implement a function `conditional_negate(int v, bool flag)` that returns `-v` if `flag` is true, else `v`. Test with positive, negative, and zero values of `v` and both flag values.
- Derive why `(v ^ -f) + f` works: start by considering binary XOR with all 1s (which inverts bits), and how adding 1 completes the two's complement negation. Write out a simple 4-bit example to confirm.
- Consider the range of `v`. Is there any input where this might not work as expected? (Hint: similar to the absolute value case – the negation of the minimum integer might overflow in signed arithmetic, but in two's complement it wraps around to itself, so the formula will just give that back.)

2.5 Merging Bits from Two Values According to a Mask

What it does: Combines two values `a` and `b` into a result `r`, by selecting bits from either `a` or `b` depending on a mask `m`. This is like a *bitwise mix*: for each bit position, if the mask has a 1, take the bit from `b`; if the mask has 0, take it from `a`. In other words, `r = (a & ~m) | (b & m)`.

How it works (bit-hack version): A compact formula for this is:

```
r = a ^ ((a ^ b) & m);
```

This works because `(a ^ b)` gives 1s for every bit that differs between `a` and `b`. Masking that with `m` means "the bits that differ *and* we choose from `b`" (since `m` has 1 where we want `b`'s bits). XORing this with `a` will flip those bits in `a` that we wanted to take from `b`, effectively inserting `b`'s bits at those positions.

Example – C Implementation:

```c
unsigned int a = 0xAAAA;   // 1010 1010 1010 1010 in binary
unsigned int b = 0xF0F0;   // 1111 0000 1111 0000 in binary
unsigned int m = 0x0F0F;   // mask: 0000 1111 0000 1111
unsigned int r = a ^ ((a ^ b) & m);
printf("Merged result = 0x%X\n", r);
```

Let's manually verify one bit group: mask `m` has bits 0-3 and 8-11 set to 1, meaning we want those bits from `b`, and the rest from `a`. Indeed, if you compute `r` using the formula or the definition `(a & ~m)|(b & m)`, you'll find `r` has `b`'s low 4 bits and `b`'s bits 8-11, while all other bits from `a`. The XOR formula does this in one line. For our sample values, `r` should come out as `0xA0FA` (you can check this as an exercise).

Use Cases: This operation is essentially a *bit-level blend* of two values, controlled by a mask. It's useful in graphics (e.g., blending two bitfields), data packing/unpacking, or implementing certain logical operations. CPUs even have vector instructions (like *blendv* in SSE) to do similar operations on a larger scale. In software, this trick achieves the blend without branching.

Practice Exercises:

- Write a generic function `merge_bits(unsigned a, unsigned b, unsigned mask)` that implements the above. Test with random masks and values against a simple loop that sets each bit of the result by checking the mask bit.
- If the mask is all 1s, what is the result? If mask is all 0s? Check that the formula reduces to those expected outcomes (should yield `b` for all 1s mask, `a` for all 0s mask).
- Consider a real-world scenario: you have two 32-bit packed pixel values and a mask for a particular channel, how could this be used to merge channels from two pixels?

Day 2 Summary: We eliminated branches in arithmetic and bit selection operations: computed absolute values and mins/max, and performed conditional bit set/clear/negation and bit merging – all using bitwise operations and arithmetic tricks. The key theme was using masks (all bits 0 or 1) derived from conditions to choose results instead of `if` statements. These patterns are fundamental in writing high-speed low-level code. As you practice, pay attention to the binary patterns and verify each step, possibly printing intermediate values in binary for clarity. Next, we will move on to techniques for counting and analyzing bits in an integer.

Day 3: Counting Bits and Parity

Topics: Efficiently counting the number of 1-bits in an integer (population count) using various methods, and determining the parity (odd or even number of 1-bits). We'll cover multiple approaches, from basic loops to clever parallel algorithms and look-up tables. Understanding these will deepen your bit manipulation skills.

3.1 Counting Bits Set to 1 – Naive Method

What it does: Counts how many bits in an integer `v` are 1 (also called the Hamming weight or population count).

How it works: The simplest method is to loop through all bit positions, incrementing a count whenever the least significant bit is 1, then shifting the number right until it becomes 0. In pseudocode:

```
count = 0
while(v != 0):
    count += (v & 1)
    v >>= 1
```

This method checks each bit one by one.

Example – C Implementation (naive):

```c
unsigned int v = /* input value */;
unsigned int c = 0;
while(v) {
    c += v & 1;    // add the lowest bit
    v >>= 1;       // shift right by 1
}
printf("Bit count = %u\n", c);
```

This will correctly count bits. Its runtime is proportional to the number of bits in `v` (for a 32-bit int, up to 32 iterations). If `v` is large (e.g., 64-bit or if used repetitively), this might be slow.

Use Cases: Bit counting is ubiquitous (e.g., calculating population counts for bitboards in chess, checking how many flags are set, cryptography, etc.). The naive method is fine for occasional use but not optimal for heavy use.

Practice: Try implementing this in Python and verify it against Python's built-in `bin(x).count("1")` for random numbers. Measure how performance scales with the size of `v` (number of bits set doesn't matter for this method, only total bit-width).

3.2 Counting Bits – Brian Kernighan's Method

What it does: A faster bit count that loops only as many times as there are 1-bits, by repeatedly turning off the rightmost set bit.

How it works: Brian Kernighan's algorithm uses the trick `v = v & (v - 1)` to clear the lowest-set 1 bit in each iteration. This works because subtracting 1 flips all bits from the rightmost 1 (inclusive) – thus `v & (v-1)` drops that lowest 1. We count how many times we can do this until `v` becomes 0.

Example – C Implementation (Kernighan):

c

```c
unsigned int v = /* input */;
unsigned int c = 0;
for (c = 0; v; ++c) {
    v &= (v - 1);  // clear the lowest set bit
}
printf("Bit count = %u\n", c);
```

Each loop iteration removes one 1-bit from `v`, so the number of iterations equals the number of set bits. In the best case (v=0) it runs 0 times, in the worst case (v has all bits =1) it runs as many times as bit-width (e.g., 32 times for 0xFFFFFFFF). On average, for random data, it runs a number of iterations equal to the population count (which is on average half the bits, if assuming random 0/1 distribution, ~16 for 32-bit).

Why it's faster: If `v` has only a few set bits (sparse bit population), this is much faster than looping through every bit. Even if `v` has many bits, it still tends to outperform the naive method due to fewer loop overheads in typical scenarios.

Use Cases: This algorithm is a common idiom in programming interviews and competitive programming for bit counting. It's efficient and the code is succinct.

Practice: Implement Kernighan's method and test it. Try it on inputs with known numbers of bits set (e.g., powers of 2 have 1 bit, all-bits-set has N bits, alternating bits, etc.) to ensure it gives correct counts. Explain why `v & (v-1)` removes the lowest set bit (trace a binary example, e.g., `v = 01101000, v-1 = 01100111, v & (v-1) = 01100000`, indeed the lowest 1-bit at position 4 was removed).

3.3 Counting Bits – Lookup Table Method

What it does: Uses a precomputed table of bit counts for smaller chunks (like 8-bit bytes), then sums up results for each chunk of the number. This trades memory for speed by doing table lookups instead of bit operations.

How it works: We allocate an array `BitsSetTable256[256]` where each entry [0..255] stores the number of 1s in that 8-bit value. This table can be filled once (e.g., using a small loop or initialization). Then for a 32-bit value, break it into four bytes and sum the table values:

```
c = table[v & 0xFF]
  + table[(v >> 8) & 0xFF]
  + table[(v >> 16) & 0xFF]
  + table[(v >> 24) & 0xFF];
```

Each `& 0xFF` extracts one byte of `v`. This yields the total count.

Example – C Implementation (lookup table):

```c
static unsigned char BitsSetTable256[256];
static void initTable() {
    BitsSetTable256[0] = 0;
    for(int i = 1; i < 256; ++i) {
        BitsSetTable256[i] = (i & 1) + BitsSetTable256[i >> 1];
    }
}
// ... (call initTable() once at program start)

unsigned int v = /* input */;
unsigned int c = BitsSetTable256[v & 0xFF]
               + BitsSetTable256[(v >> 8) & 0xFF]
               + BitsSetTable256[(v >> 16) & 0xFF]
               + BitsSetTable256[(v >> 24) & 0xFF];
printf("Bit count = %u\n", c);
```

We fill the table such that each entry is the count of bits in the index (this can also be done with a compile-time constant initializer). Then we do 4 lookups and 3 additions to get the result for 32-bit `v`. This is a fixed amount of work, independent of how many bits are in `v`.

Use Cases: Lookup tables are great when you need constant-time operations and can afford the memory. Counting bits via table is extremely fast because it uses O(1) operations for a fixed size (for 32 bits, 4 lookups). It's used in optimized libraries or when processing a stream of values (where the table remains in cache, making lookups very fast).

Pitfalls: The table uses 256 bytes, which is trivial, but if you extended it to 16-bit chunks you'd need 65536 entries which is larger (not usually worth it). 8-bit chunks hit a sweet spot. Also ensure the table is initialized before use (common mistake is forgetting to call the init and getting wrong counts).

Practice: Implement the above and test it. Compare its speed to the previous methods by running it on a large array of random numbers and measuring time (if you have access to timing). Check that it returns the same counts as the other methods for various inputs.

3.4 Counting Bits – "Parallel" Bit Count (HW-inspired)

What it does: Uses a series of arithmetic operations to count bits in parallel across the word, combining partial counts. This is a rhythm often used in bit-level hacks and also found in Henry Warren's *Hacker's Delight*. It yields the count in a logarithmic number of steps by manipulating bit groups.

How it works: The idea is to successively combine bits into groups and count within those groups using clever bit masks and addition. One known sequence for 32-bit integers is:

```c
v = v - ((v >> 1) & 0x55555555);                    // pair bits count
v = (v & 0x33333333) + ((v >> 2) & 0x33333333); // nibble count
v = (v + (v >> 4)) & 0x0F0F0F0F;                    // byte count
int c = (v * 0x01010101) >> 24;                     // multiply sum and extract
```

Breaking it down:

- `0x55555555` in binary is `01010101...` (repeating). `(v >> 1) & 0x55555555` shifts right and masks to keep only even-position bits of original. `v - (...)` subtracts that from original `v`. This essentially computes partial counts for each pair of bits. After this, each 2-bit group of `v` holds the number of 1s in those two original bits.
- Next mask `0x33333333` is `00110011...`. The code then separates the 2-bit groups and sums them into 4-bit (nibble) groups: `(v & 0x33333333)` keeps bits 0-1 of each pair, `((v >> 2) & 0x33333333)` gets bits 2-3 of each pair, then adds. Now each 4-bit nibble contains the count of ones in the original 4-bit nibble.
- `(v + (v >> 4)) & 0x0F0F0F0F` adds neighboring nibble counts, giving counts in each 8-bit byte (mask `0x0F0F0F0F` = `00001111...` extracts lower 4 bits of each sum which now hold up to 8, which fits in 4 bits).
- Finally, multiplying by `0x01010101` and shifting right 24 effectively adds the counts of all 4 bytes together. Why multiplication? When we multiply the 8-bit counts by this constant, it replicates the sum across higher bytes due to how multiplication works with those patterns (this uses the fact that `0x01010101 * x` will produce a result where each byte is added into the most significant byte). After this multiplication, the high byte of the 32-bit word contains the total count, which we then shift down.

Example – C Implementation (parallel count):

```c
unsigned int v = /* input */;
v = v - ((v >> 1) & 0x55555555);
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
v = (v + (v >> 4)) & 0x0F0F0F0F;
unsigned int c = (v * 0x01010101) >> 24;
printf("Bit count = %u\n", c);
```

This yields the number of 1 bits in `v` using a fixed, small number of operations (no loops). It's effectively doing many operations in parallel across the bits (hence the name).

Use Cases: This is a classic hack for bit counting when you really want to avoid loops. It's beneficial if you expect values with lots of bits (so that even Kernighan's might loop many times) or in a setting where branchless deterministic execution time is desired. Some processors have a single instruction for popcount nowadays, but this method was commonly used before those existed (and is still useful in languages that lack direct builtins).

Practice: Step through the algorithm with a specific number. For instance, take `v = 0xF0A5` and manually execute each step to see how the bit counts accumulate in each stage. Ensure the final answer matches the actual bit count (you can cross-check with a simpler method). Try to modify the masks and steps for a 64-bit version (you'd extend the idea with more steps). If you have access to builtins like `__builtin_popcount` in GCC or `_mm_popcnt_u32` (intrinsic for x86), compare the results and possibly performance with this manual method.

3.5 Additional Counting Tricks – Higher Precision and "Rank" (Advanced)

(Optional advanced content, can be treated as extra practice for the curious.) The Bit Twiddling Hacks page also includes routines for counting bits in specific bit-lengths using 64-bit operations, and even finding the count of bits up to a certain position ("rank") or selecting the nth set bit. These go beyond basic popcount, but are applications of the above methods:

- **Counting bits in 14, 24, or 32-bit words using 64-bit instructions:** This trick packs multiple smaller counts into one 64-bit register to do two counts at once, leveraging 64-bit arithmetic to process, say, two 32-bit values in parallel. It's an optimization that might be used in systems where 64-bit multipliers are available to speed up counting multiple values.
- **Rank and Select operations:** *Rank* is the number of 1-bits up to a given position. The hacks combine bit-parallel operations similar to the above to compute rank efficiently. *Select* finds the position of the Nth 1-bit. These are quite complex but rely on the same principles of splitting and combining counts, plus some additional logic. They're used in advanced data structures (bit vectors, wavelet trees, etc.).

While delving into these is beyond our one-week scope, it's good to know they exist as an extension of bit counting. If interested, refer to the "Count bits set (rank)…" and "Select bit position with given rank" sections on the Bit Twiddling Hacks page.

Practice (advanced): If you're curious, try implementing a function that given a 64-bit integer and a position `pos`, returns the number of 1s up to that `pos` (rank). Use the parallel count method to get the total count in parts of the number, and then mask off bits after `pos`. Conversely, think about how you might find the position of the k-th 1-bit by using successive refinements (hint: you could binary search on bit groups using the counts technique).

Day 3 Summary: We learned multiple techniques to count bits – from a simple loop to using clever arithmetic and tables to greatly speed it up – and also how to determine parity (which is essentially count bits mod 2, see below). In doing so, you got exposed to the idea of processing bits in parallel (combining them into groups) and using mathematical operations to replace loops. You should now be comfortable implementing a popcount in more than one way and understand the trade-offs (e.g., code size and complexity vs. speed, and when branches might be slow vs. when a lookup table is beneficial). In the next day, we will tackle bit tricks for rearranging bits: swapping and reversing bit sequences.

Day 4: Swapping and Reversing Bits

Topics: Swapping two variables without a temporary, swapping individual bits within a number, and reversing the bit order in a binary number using various methods (loop, lookup table, and parallel operations). These operations rewire the bits and are classic interview questions and hardware-level routines.

4.1 Swapping Two Variables with Addition/Subtraction

What it does: Exchanges the values of two variables `x` and `y` without using an extra temporary variable, by leveraging arithmetic.

How it works: The idea is to use addition and subtraction (or XOR, in the next hack) so that the two variables sort of "hold" each other's values during the swap:

```
x = x + y;
y = x - y;
x = x - y;
```

After the first line, `x` holds the sum of original `x` and `y`. Then we subtract the new `y` (which is still original `y`) from `x`'s sum to get original `x` back into `y`. Finally, subtract the new `y` (original x) from the sum to get original `y` into `x`. This works because the operations are reversible.

Example – C Implementation (add/sub swap):

```c
int x = 5, y = 9;
x = x + y;  // x = 14, y = 9
y = x - y;  // y = 14 - 9 = 5 (y now holds original x)
x = x - y;  // x = 14 - 5 = 9 (x now holds original y)
printf("x=%d, y=%d\n", x, y);  // outputs: x=9, y=5
```

No extra variable was used, and we successfully swapped values.

Pitfalls: This method can overflow if `x + y` exceeds the range of the integer type. In C, signed overflow is undefined behavior, so this can be dangerous if you don't control the values. For unsigned integers, it wraps around modulo arithmetic (so it's defined, but you might still not want an overflow). Thus, while academically interesting, it's safer to use XOR for general integer types (next section) or just use a temporary in real code unless you *know* the range.

Use Cases: Swapping without a temp might save space in extremely low-level contexts or might be used in code-golf. In practical terms, compilers nowadays will optimize a three-line swap into an optimal sequence anyway (and often using a register as temp is fine). So consider this a neat trick more than a go-to for everyday programming.

Practice: Swap two numbers using this method and also try it on paper with some values to be sure it works. Test what happens if you try it with extremely large values in a language like Python (which can handle big ints) to simulate overflow and see the effects.

4.2 Swapping Two Variables with XOR

What it does: Another temp-less swap using bitwise XOR:

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

This achieves the swap without the potential overflow issue of addition.

How it works: XOR has the property that if you XOR a value with the same value again, you get 0, and if you XOR with 0, you get the original value. The sequence works as follows:

- After `x = x ^ y`, `x` holds `orig_x ^ orig_y`, and `y` is still `orig_y`.
- Next `y = x ^ y` becomes `orig_x ^ orig_y ^ orig_y`. Since `orig_y ^ orig_y = 0`, this simplifies to `orig_x`. So now `y` has `orig_x`.
- Finally `x = x ^ y` becomes `(orig_x ^ orig_y) ^ orig_x`. XOR is associative and commutative, so that equals `orig_y` (because `orig_x ^ orig_x = 0` leaves `orig_y`). Now `x` has `orig_y`. Swap done!

Example – C Implementation (XOR swap):

c

```c
int x = 5, y = 9;
x = x ^ y;
y = x ^ y;
x = x ^ y;
printf("x=%d, y=%d\n", x, y);  // outputs: x=9, y=5
```

This should output the values swapped, just like the addition method. No overflow concerns here, since XOR just operates on bits.

Pitfalls: There is one edge case: if `x` and `y` refer to the same memory location (e.g., if you do `swap(x, x)`), this will zero out `x`. In practice, if you're swapping distinct variables, that won't happen. But it's a gotcha if someone wrote a macro or function that could possibly be called with the same variable for both arguments. Using a temp variable in that case would do nothing (which is correct behavior: swapping the same thing leaves it unchanged), whereas the XOR swap would corrupt the data. So ensure the addresses are different (which is normally true when you explicitly write `x` and `y`).

Use Cases: Same as above – mostly educational. In performance terms, XOR swap might or might not be any better than using a temp (and can be worse if the CPU can't parallelize XORs as well as it can do a simultaneous register renaming for a normal swap). Modern compilers might recognize this pattern and handle it optimally though.

Practice: Implement XOR swap and test it on different data types (e.g., try with `char` or `long` to show it's type-agnostic as long as the types are the same and bitwidth matches). Also, explain step-by-step using a concrete binary example how the bits move. For instance, x=6 (0110), y=9 (1001):

- After first XOR: x = 1111, y = 1001.
- After second XOR: x = 1111, y = 0110.
- After third XOR: x = 1001, y = 0110 – swapped.

4.3 Swapping Individual Bits with XOR

What it does: Swaps two specific bit positions within a single number (or across two numbers) using XOR. For example, swap bit i and bit j in integer `v`.

How it works: The general approach:

1. Compute a mask of the two bits: `mask = ((v >> i) ^ (v >> j)) & 1`. Actually, that computes whether the bits differ. A more direct method: `mask = ((v >> i) & 1) ^ ((v >> j) & 1)` (this is 1 if the bits are different, 0 if the same).

2. If the bits are the same, no need to swap. If they are different (mask = 1), we can flip both bits by doing: `v ^= (1U << i) | (1U << j)`.

A concise formula given in the Bit Hacks collection is:

```
unsigned int bitMask = (1U << i) | (1U << j);
v ^= bitMask;
```

provided we know the bits differ; otherwise, doing XOR will flip them (if they differ) or do nothing (if they're the same) – actually that covers both cases correctly without needing an explicit check because if the bits are the same, flipping both will still keep them same (just both flipped from 0 to 1 or 1 to 0, which is not what we want for a swap – oops, careful: if bits are same and we flip both, we haven't swapped, we've changed the value!). So we should ensure we only flip if they differ. A safe approach is:

```c
if (((v >> i) & 1) != ((v >> j) & 1)) {
    unsigned int bitMask = (1U << i) | (1U << j);
    v ^= bitMask;
}
```

This will swap the bits.

Example – C Implementation (swap two bits in a single int):

```c
unsigned int v = 0b11010010;  // for demonstration (8-bit example)
unsigned int rev = 0;
for(int i = 0; i < 8; ++i) {
    rev <<= 1;                // make room for next bit
    rev |= (v >> i) & 1;      // copy LSB of (v >> i) into rev
}
printf("Reversed = 0b%08b\n", rev);
```

If `v = 11010010` (which is 0xD2, just as an example), the reversed should be `01001011` (0x4B). The loop will achieve that. For 32-bit, the loop runs 32 iterations.

Use Cases: Reversing bits is sometimes needed in algorithms (for example, certain DSP algorithms, Fast Fourier Transform bit-reversal permutations, generating Gray codes, etc.).

Complexity: O(N) for N-bit word, which for N=32 is 32 operations – not bad, but we can do better as we'll see.

Practice: Implement a general bit-reverse for a 32-bit `unsigned`. Test it on known values:

- The reverse of `0xFFFFFFFE` (all 1s except last bit) should be `0x7FFFFFFF` (all 1s except first bit).
- The reverse of a palindrome bit-pattern (like 0x0F0F0F0F which is symmetric in 4-bit blocks but not overall symmetric, maybe a better example: 0xF0F00F0F) – just to see it works in all cases. Compare the output of your function with a brute-force Python approach for random inputs to verify.

4.5 Reversing Bits – Lookup Table Method

What it does: Similar to counting bits via table, we can reverse bits using a lookup table for smaller chunks (e.g., reverse of all 8-bit numbers precomputed), then assemble for a larger word.

How it works: Precompute an array `ReverseTable256[256]` where each index is 0–255 and the value is the 8-bit reverse of that index. To reverse a 32-bit value:

- Split it into 4 bytes.
- Look up the reverse of each byte.
- Reassemble in opposite order: the reverse of the lowest byte becomes the highest byte of the result, etc.

In code:

```
rev = (ReverseTable256[v & 0xFF] << 24) |
      (ReverseTable256[(v >> 8) & 0xFF] << 16) |
      (ReverseTable256[(v >> 16) & 0xFF] << 8) |
      (ReverseTable256[(v >> 24) & 0xFF]);
```

This gives the 32-bit reversed result.

Example – C Implementation (using a table):

First, fill the reverse table for 8-bit values (you can do this by looping 0 to 255 and using the loop method for 8 bits, or by clever bit math). Then:

c

```c
unsigned int v = /* 32-bit value */;
unsigned int b0 = (v & 0xFF);
unsigned int b1 = (v >> 8) & 0xFF;
unsigned int b2 = (v >> 16) & 0xFF;
unsigned int b3 = (v >> 24) & 0xFF;
unsigned int rev = (ReverseTable256[b0] << 24) |
                   (ReverseTable256[b1] << 16) |
                   (ReverseTable256[b2] << 8)  |
                    ReverseTable256[b3];
```

This does 4 table lookups and a few shifts/or, which is constant time relative to word size (assuming word size is fixed to 32 or 64 bits).

Use Cases: Same as bit reversal in general, but this method is faster when you need to reverse many numbers frequently. The table of 256 bytes is small and likely stays in cache.

Practice: Implement the table and the reversal using it. Test on some values and compare with your loop method to ensure it matches. If you're adventurous, try to extend to 16-bit table (you'd need 65536 entries, which is 64KB, somewhat large but doable) and then you could reverse 32-bit with 2 lookups instead of 4 (though 4 lookups of a smaller table might be faster due to cache). Evaluate the trade-off.

4.6 Reversing Bits – 64-bit Multiply Method (and Variations)

What it does: Surprising as it sounds, we can use multiplication to help reverse bits! There are known constants to use with 64-bit multiplication to achieve bit reversal of bytes. Specifically, the hack uses a multiplication and a modulus or a shift to spread bits.

One method from the Bit Twiddling Hacks uses 64-bit operations to reverse bits in a byte in only **3 operations**:

c

```c
unsigned char b = /* 8-bit value */;
unsigned long long m = (unsigned long long) b * 0x0202020202ULL;
unsigned long long mask = 0x010884422010ULL;
unsigned long long shifted = m & mask;
unsigned char reversed_b = (unsigned int)(shifted * 0x010101010101 >> 32);
```

This is quite complex, but essentially:

- Multiply the byte by a magic number that spreads its bits across a 64-bit value with gaps.
- Mask with another magic constant to isolate those bits in proper positions.

- Multiply by another magic constant to combine them back into the byte reversed.

The specifics of those constants are beyond a quick intuitive explanation (they come from analyzing the binary pattern needed to align bits). The result `reversed_b` is the reversed 8-bit. You can then do similar assembly as before to reverse a full 32-bit by treating it as four bytes reversed (or use 64-bit registers to do even 32 at a time).

There are variations that avoid actual division/mod but use shifts. For example, a sequence of shifts and masks (the "parallel" approach to bit reversal) for 32-bit:

```c
v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);
v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);
v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);
v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);
v = ( v >> 16             ) | ( v               << 16);
```

This performs swaps in a divide-and-conquer: first swap every 1-bit pair, then 2-bit pairs, then nibbles, then bytes, then 16-bit halves. At the end, `v` is reversed. This is 5 * log2(32) = 5*5 =25 operations, which is branchless and still quite fast.

Example – C Implementation (parallel reversal using masks):

```c
unsigned int v = /* input */;
// Swap odd/even bits
v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);
// Swap pairs
v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);
// Swap nibbles
v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);
// Swap bytes
v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);
// Swap 16-bit halves
v = (v >> 16) | (v << 16);
```

After this, `v` is bit-reversed. Each line is swapping progressively larger bit groups.

Use Cases: This method is good when you want to avoid a loop and maybe work in a fixed number of operations. It might be overkill for one-off, but if you had to reverse billions of 32-bit words, using this or the LUT method would be wise.

Practice: Implement the parallel bit reversal as above and test on a range of values to ensure correctness. Compare the performance (if possible) between the simple loop vs lookup vs parallel method by running each many times. Also consider code size vs speed: the parallel method is a few lines of code but quite readable if you know the constants; the LUT requires setting up the table; the loop is smallest in code but potentially slower.

Day 4 Summary: We explored swapping techniques (including the famous XOR swap) and various ways to reverse bits in a value. We saw how bit-level operations can avoid extra memory usage (swaps without temp) and how reversing can be optimized using tables or parallel operations. A recurring theme is using masks to isolate parts of the number and operate on them (for swapping bit pairs or reversing chunks). With these, you're getting more comfortable manipulating bit patterns at will. Tomorrow, we'll move into techniques for finding specific bit positions (like computing logarithms base 2, finding trailing zeros) and dealing with powers of two.

Day 5: Bit Scanning – Finding Bit Positions and Rounding

Topics: Finding the position of the highest-set bit (integer log2), finding the count of trailing zero bits, and rounding up to the next power of 2. These hacks allow us to analyze the bit layout of a number (essentially "bit scanning" from MSB or LSB side) without loops or with minimal loops.

5.1 Integer Log2 (Position of Highest Set Bit) – Iterative and Binary Search

What it does: Finds the index of the most significant set bit (MSB) in an integer. This is effectively the floor of log2(v). For example, if `v=20 (10100)_2`, the highest set bit is at position 4 (0-indexed from LSB, or it's the 5th bit from the right), which corresponds to floor(log2(20)) = 4.

Method 1: Simple loop: Right-shift `v` until it becomes 0, count how many shifts were done:

```c
int idx = -1;
while(v) {
   v >>= 1;
   idx++;
}
```

After this, `idx` will be the index of the last bit removed (i.e., MSB index). This is O(N) in the number of bits (up to 31 shifts for 32-bit).

Method 2: Binary search (branching): We can check halves of the bits at a time using if/else:

c

```c
int idx = 0;
if(v >> 16) { v >>= 16; idx += 16; }
if(v >> 8)  { v >>= 8;  idx += 8; }
if(v >> 4)  { v >>= 4;  idx += 4; }
if(v >> 2)  { v >>= 2;  idx += 2; }
if(v >> 1)  { /* no shift needed */ idx += 1; }
```

This effectively jumps in powers of two. For a 32-bit, after these checks, `idx` will be floor(log2) (because each step we cut down the range where the MSB could be by seeing if the upper half is non-zero, etc.). This method uses a few branches but only a fixed small number (5 checks for 32-bit).

Method 3: Use built-ins or bit hacks: Modern compilers have builtins like `__builtin_clz` (count leading zeros) which directly give this. Or one can use a de Bruijn sequence trick (an advanced method) to find it in a parallel way using multiplication, which is mentioned in some bit hack discussions, but binary search is simpler to grasp.

Example – C Implementation (binary search method):

c

```c
unsigned int v = /* input */;
int idx = -1;
if(v == 0) {
    idx = -1; // no set bits (define as -1 or some flag)
} else {
    idx = 0;
    if(v >> 16) { v >>= 16; idx += 16; }
    if(v >> 8)  { v >>= 8;  idx += 8; }
    if(v >> 4)  { v >>= 4;  idx += 4; }
    if(v >> 2)  { v >>= 2;  idx += 2; }
    if(v >> 1)  { /* v >>= 1; */ idx += 1; }
    // idx now is the position (0-based) of MSB
}
printf("Highest set bit index = %d\n", idx);
```

Test: if v = 20 (10100b), after `if(v>>16)` false (since v is 20, >>16 =0), next `if(v>>8)` false (20>>8=0), `if(v>>4)` true (20>>4 = 1 which is non-zero), so v becomes 1, idx +=4. Next `if(v>>2)` false (1>>2=0), `if(v>>1)` false (1>>1=0). End with idx=4, which is correct. If v were 0, idx stays -1 indicating no bits set.

Use Cases: Finding MSB is useful for determining bit-width of a number, or for algorithms like computing logarithms, or formatting, etc. Many algorithms choose different strategies based on the magnitude of numbers (often depending on highest bit).

Pitfalls: If `v` is 0, there is no set bit, so you need to decide how to handle that (here we used -1 to indicate "none"). Also, ensure the data types and shifts cover the full range (the above assumes 32-bit, if using 64-bit you'd extend the pattern).

Practice: Implement both the simple loop and the binary search approach and test on various values (especially edge cases like 0, powers of 2 (which should return their exponents), and all 1s (should return N-1 as index)). Also, check with an example like v = 1 (should get 0), v = 2 (1), v = 3 (1, since highest bit is the 2^1 place), etc.

5.2 Integer Log2 – Using Floating Point Hack (advanced variant)

(This is an optional curiosity): The Bit Twiddling Hacks page mentions finding log2 using floating-point representation. By interpreting the bits of a float, one can extract the exponent which gives log2 for powers of 2 and interpolate. For example, for a 32-bit float, if you cast the int to float, the exponent field (after bias) essentially gives floor(log2). However, this requires careful handling and is architecture-dependent (and not as practical in modern code because of potential undefined behavior of aliasing). It's a neat trick but beyond our main discussion.

If interested: `int log2val; float f = (float)v; log2val = ((*(int*)&f) >> 23) - 127;` would yield floor(log2(v)) for v up to 2^24 (and needs adjustments beyond). The page provides more robust variants.

5.3 Counting Trailing Zero Bits – Loop and Parallel Methods

What it does: Counts how many consecutive 0 bits starting from the least significant bit (LSB) of `v`. For example, `v = 40 (101000)_2` has 3 trailing zeros (the binary ends in `000`). If `v` is even, there is at least 1 trailing zero (because it's divisible by 2). If `v` is odd, trailing zero count is 0 (because it ends in 1).

Method 1: Loop (linear scan): Continuously check `v & 1` and shift right until you hit a 1 or run out of bits:

```c
unsigned int v = /* input */;
int count = 0;
while((v & 1) == 0 && count < 32) {
    count++;
    v >>= 1;
}
```

This examines each bit from LSB upward until a 1 is found. If `v` becomes 0, you might define count as 32 (meaning all bits were 0, if you treat trailing zeros of 0 as 32 for 32-bit).

Method 2: Using bit hack with `(v & -v)`: A known trick: `v & -v` isolates the lowest set bit of `v` (it gives a value that has only that one bit, all lower bits zero). Example: `v=40 (101000)_2, -v = 2's complement = (~101000)+1 = (010111)+1=011000 (which is -40 mod 2^n)`, `v & -v = 001000` which is 8, the lowest set bit. Now, if we take logs or use a lookup on this value, we can find the position of that bit which equals the count of trailing zeros.

One approach: use a lookup table for `v & -v` values. But an elegant approach uses a multiply with a de Bruijn sequence (a technique beyond scope for detail) to map the isolated bit to an index in a small number of operations.

Alternatively, the Bit Hacks suggests a parallel method:

```c
// Approach: Flip trailing zeros to 1s, then count
if(v) {
    int c = 0;
    v = (v ^ (v - 1)) >> 1;  // This sets all trailing 0s to 1, and others to 0
    // Now just count bits in v (which equals count of trailing zeros originally)
    while(v) { v >>= 1; c++; }
    // c is the count of trailing zeros
} else {
    // define c = word size if v is 0, as it has all bits zero
}
```

The line `v = (v ^ (v - 1)) >> 1` deserves explanation: `(v - 1)` will have all bits at and below the lowest set bit of `v` flipped. XORing that with `v` thus gives a mask that has 1s in all those flipped positions (which correspond to the trailing zeros and the lowest set bit of original). By shifting right 1, we drop the lowest set bit's marker, leaving 1s only for each trailing zero originally. Then counting those 1s (which we know how to do from Day 3) gives the trailing zero count. This is a bit roundabout but avoids looping through each bit individually if there are many trailing zeros.

Method 3: Builtin or CPU instruction: Many CPUs have an instruction to count trailing zeros (e.g., x86 `BSF` instruction). Compilers often expose these as builtins like `__builtin_ctz` in GCC. If allowed, that's simplest and fastest: `count = __builtin_ctz(v)` (with the caveat that if v is 0, __builtin_ctz is undefined, so handle that separately).

Example – C Implementation (using de Bruijn, a known 32-bit sequence trick):

The de Bruijn sequence approach uses a magic 32-bit number `0x077CB531` and a lookup table of 32 entries. Due to complexity, we'll not derive it fully here, but typically:

c

```c
static int index32[32] = {0,1,28,2,29,14, // ... initialize with a specific permutatio
int ctz32(uint32_t v) {
    return index32[((uint32_t)((v & -v) * 0x077CB531)) >> 27];
}
```

This works by mapping the isolated bit through a perfect hash to a precomputed index. It's constant time.

For understanding, using the simpler approach:

c

```c
unsigned int v = /* input */;
if(v == 0) { count = 32; }
else {
    unsigned int w = (v ^ (v - 1)) >> 1;  // isolate trailing ones of ~v
    // now count bits in w (popcount)
    w = w - ((w >> 1) & 0x55555555);
    w = (w & 0x33333333) + ((w >> 2) & 0x33333333);
    w = (w + (w >> 4)) & 0x0F0F0F0F;
    count = (w * 0x01010101) >> 24;
}
```

This yields the same count without looping.

Use Cases: Trailing zero count is used for tasks like finding the factor of 2 in a number's factorization (how many times it's divisible by 2), or finding alignment (e.g., if you have an address and want to know how aligned it is, count trailing zeros). Also used in bitboard operations in chess programming (to get index of least significant set bit quickly).

Practice: Implement `count_trailing_zeros(unsigned v)` using both a simple loop and one of the above faster methods. Test it on various values: powers of 2 (which have trailing zeros equal to their exponent, e.g., 8 (1000b) has 3), odd numbers (should return 0), 0 (decide on return value, e.g., 32 for 32-bit). Compare the results with say a brute-force Python method (e.g., find index of first '1' in bin(v) from right).

5.4 Rounding Up to Next Power of 2

What it does: Given a number `v`, find the smallest power of 2 that is >= `v`. This is often used in allocating array sizes, hashing, etc., when you want a power-of-two size at least as large as a given number.

How it works: A common idiom (assuming 32-bit, and v > 0) is to "fill all bits to the right of the MSB with 1s, then add 1". The steps:

- `v--;` (decrement v first, to handle the case when `v` is already a power of 2 – in that case we would otherwise go to the *next* power, but we want the same number).
- `v |= v >> 1;`
- `v |= v >> 2;`
- `v |= v >> 4;`
- `v |= v >> 8;`
- `v |= v >> 16;`
- `v++;`

This sequence sets all bits below the highest set bit to 1. After that, adding 1 will carry out through those ones and result in 1 followed by all 0s – which is the next power of two above the original. If the original `v` was itself a power of 2, the decrement at the start ensures we don't overshoot.

Example – C Implementation:

```c
uint32_t v = /* input */;
// Handle v=0 as a special case if you want (often round up of 0 is 1 or 0 depending o
if(v == 0) {
    nextPow2 = 1;
} else {
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    v++;
    uint32_t nextPow2 = v;
    printf("Next power of 2 = %u\n", nextPow2);
}
```

Test this: say `v=13 (1101)_2`. Decrement to 12 (1100). OR with 12>>1=6 (0110) gives `1100 | 0110 = 1110`. OR with >>2 (0011) gives `1111`. OR with >>4 (0000) stays `1111`. etc. Now add 1 gives `10000` (16) which is correct. If `v` was exactly 16, then after decrement it becomes 15 (01111), filling steps make it 15, add 1 -> 16, so it remains 16 as desired.

Alternative method: Using the MSB index we found earlier, one could simply compute `nextPow2 = 1 << (idx+1)` unless `v` is exactly a power of 2, in which case it's `1<<idx`. But the above method does all in a branchless way.

Use Cases: Memory allocation (lots of allocators round up to pow2 for bucket sizes), algorithms that need matrix sizes power of 2, graphics (textures often using power of 2 dimensions historically), etc.

Pitfalls: If `v` is 0, the formula as given will produce 0 after decrement wraparound (for unsigned) or weird result for signed. So handle 0 separately (decide that next power of two of 0 is 1, by convention). Also, watch out if you use types bigger than the pattern of shifts (for 64-bit you'd extend more shifts).

Practice: Implement the rounding logic and test it on boundary conditions: 0 (decide output), 1 (should stay 1), 2 (stay 2), 3 (go to 4), 4 (stay 4), 5..7 (go to 8), 8 (stay 8), 9 (go 16), etc., and also a number just below 2^n. Maybe test on random numbers and verify by a simple loop that multiplies 2's until >=v.

Day 5 Summary: We focused on bit scanning: identifying positions of bits from either end of the number. You learned how to compute log2 by shifting or binary search, how to count trailing zeros efficiently, and a powerful idiom for rounding up to the next power of 2. These operations often underlie low-level optimizations in things like memory allocation and bitboard manipulations. With these, we have completed a toolbox for analyzing and adjusting bit patterns. Tomorrow, we'll explore some advanced and specialized hacks involving interleaving bits (Morton codes) and byte-level operations within words, plus the trick to generate the next combination of set bits.

Day 6: Advanced Bit Hacks – Remainders and Interleaving

Topics: Efficient computation of remainders when the divisor has special form (2^s or 2^s-1), and interleaving bits (Morton codes for spatial hashing). These are more specialized but showcase the power of bit manipulation in mathematical contexts.

6.1 Computing Remainder modulo 2^s (without division)

What it does: Calculates `v mod (1 << s)` (i.e., `v mod 2^s`) using bit operations. This one is straightforward: `2^s` is a power of two, so the remainder when dividing by it is just the lower `s` bits of `v` (because 2^s is 1 followed by s zeros in binary, and division by it chops off the lower s bits).

How it works: `v mod 2^s` can be obtained by `v & ((1U << s) - 1)`. The mask `(1<<s) - 1` produces a binary number with s lower bits = 1, higher bits = 0. ANDing with `v` zeroes out all bits except the bottom s bits, which is exactly the remainder.

Example – C Implementation:

c

```c
unsigned int v = 77;
unsigned int s = 3;
unsigned int mod = v & ((1u << s) - 1);
printf("%u mod %u = %u\n", v, (1u<<s), mod);
```

For `v=77` (1001101) and s=3 (divisor 8 which is 1000b), mask = 0x7 (111b). 77 & 7 = 5, and indeed 77 mod 8 = 5.

Use Cases: When you know your divisor is a power of two (like alignment calculations, working with sizes that are powers of two), you should use bit masking instead of slow `%` operator. Compilers often optimize `% 2^s` to this anyway, but it's good to recognize.

Practice: Just apply this in a couple of contexts: e.g., get the low 4 bits of a number (mod 16), etc. It's a simple but very commonly used trick.

6.2 Computing Remainder modulo (2^s - 1)

What it does: Calculates `v mod ((1 << s) - 1)` without actual division. For example, mod 7, 15, 31, 63, etc. This one is trickier. The constants like 3, 7, 15, 31 are all numbers with all lower bits =1. E.g., 15 (0xF) = 2^4-1.

How it works: A number like 1111 (15) has the property that adding chunks above will cycle. The trick given in Bit Twiddling Hacks for mod (2^s - 1) uses a technique of adding the upper part of the number to the lower part:

- For 32-bit mod 255 (which is 2^8-1), they do something like: break into bytes and add them, then adjust if needed.
- In general, one method: repeatedly add the higher bits to the lower bits. Specifically, for n=(1<<s)-1, one can do:

  ```
  while(v >= n) {
     v = (v >> s) + (v & n);
  }
  ```

  Because shifting right by s is dividing by 2^s, and taking `v & n` is the remainder after dividing by 2^s. By adding them, we effectively do the "digital sum" in base 2^s representation (which is analogous to how you compute mod 9 by adding decimal digits for instance).

For a concrete example, `v mod 7 (0b111)`: Suppose v is in binary and we split it into groups of 3 bits. Adding those groups together (in binary) and repeating will yield the result mod 7. This works because 2^s ≡ 1 (mod 2^s-1), so any 2^s * A + B ≡ A + B (mod 2^s-1). So you can fold the number.

The Bit Twiddling Hacks page shows a specific constant-based approach for doing mod 255 and such in parallel without a loop. For example, mod 255 can be done by:

```c
unsigned int x = (v & 0xFF) + ((v >> 8) & 0xFF) + ((v >> 16) & 0xFF) + ((v >> 24) & 0x
x = (x & 0xFF) + (x >> 8); // in case sum overflows 8 bits
if(x > 255) x -= 255;
```

This adds all bytes because 256 ≡ 1 (mod 255). Similarly for mod 15 (0xF), you'd add each nibble.

The page also describes doing it in parallel with some operations (the "in parallel" likely refers to a vectorized approach inside one number using masks and multipliers).

Example – C Implementation (mod 7 via loop method):

```c
unsigned int v = 100;
unsigned int n = 7; // (1<<3)-1
while(v >= 7) {
    v = (v >> 3) + (v & 0x7);
}
printf("mod 7 = %u\n", v);
```

For 100 (1100100b), group into 3-bit chunks: 1 100 100 (which is 1,4,4). Add: 1+4+4 = 9. Since 9 >=7, do again: 9 in binary 1001 grouped as 1 001: 1+1 = 2. 2 <7 stop. Indeed 100 mod7 = 2.

Use Cases: mod (2^s -1) appears in certain algorithms, e.g., computing checksums (like sum of bytes mod 255 for checksum), or in digital root computations, etc. It's a neat trick to avoid division especially for something like mod 9 or mod 3 in binary-coded decimal, etc. It's somewhat niche though.

Practice: Try the above method for mod7, mod15, mod255 on some numbers, including edge cases (like number just one less than the modulus, number equal to modulus which should give 0). Ensure the logic holds.

6.3 Interleaving Bits (Morton Codes)

What it does: Interleaves the bits of two (or more) numbers to form a single number. For instance, given two 16-bit coordinates x and y, produce a 32-bit Morton code where bits of x and y are interleaved: all even positions are bits of x, and all odd positions are bits of y. This is used in spatial indexing (Z-order curve).

How it works: To interleave, we need to spread the bits of a number apart by inserting zeros between them, then OR them together. For two numbers:

- Clear the original number's bits to ensure they only occupy even (or odd) positions.
- Use a method to **expand** each bit of x so that it takes 2 bits in the result (the known term is "pivoting" bits).

A method from the Bit Hacks uses magic constants and shifts:

```c
uint32_t x; // 16-bit input
uint32_t y; // 16-bit input
uint64_t morton;
uint32_t X = (x * 0x00010001) & 0xFF0000FF;
X = (X * 0x00000101) & 0x0F00F00F;
X = (X * 0x00000011) & 0xC30C30C3;
X = (X * 0x00000005) & 0x49249249; // now X has the bits of x in even positions
uint32_t Y = similar operations on y;
morton = X | (Y << 1);
```

These constants successively spread the bits. The final mask 0x49249249 is a known pattern (`010010010010...`) that corresponds to the positions of bits in a 3D Morton code (but for 2D, it's the positions for one coordinate). Actually, 0x55555555 is often used for mask of even bits, 0xAAAAAAAA for odd.

A simpler conceptual approach:

- Start with x: `x &= 0x0000FFFF;`
- `x = (x | (x << 8)) & 0x00FF00FF;` (move bits apart in 16-bit chunks)
- `x = (x | (x << 4)) & 0x0F0F0F0F;`
- `x = (x | (x << 2)) & 0x33333333;`
- `x = (x | (x << 1)) & 0x55555555;` Now x's bits are in even positions. Do the same for y, then `morton = x | (y << 1)`.

Example – C Implementation (interleave 16-bit x,y into 32-bit):

c

```c
uint32_t x = ... & 0x0000FFFF;
uint32_t y = ... & 0x0000FFFF;
x = (x | (x << 8)) & 0x00FF00FF;
x = (x | (x << 4)) & 0x0F0F0F0F;
x = (x | (x << 2)) & 0x33333333;
x = (x | (x << 1)) & 0x55555555;
y = (y | (y << 8)) & 0x00FF00FF;
y = (y | (y << 4)) & 0x0F0F0F0F;
y = (y | (y << 2)) & 0x33333333;
y = (y | (y << 1)) & 0x55555555;
uint32_t morton = x | (y << 1);
```

This will produce the Morton code with bit 0 from x as bit0, bit0 from y as bit1, bit1 from x as bit2, bit1 from y as bit3, etc.

Use Cases: Spatial data structures (like quadtrees, octrees – Z-order curve), mapping 2D coordinates to 1D while preserving locality. Morton codes are used in GPU texture mapping (like swizzling textures in memory), and in databases for spatial indexing.

Practice: Write a function to interleave bits of two 16-bit numbers into one 32-bit. Test on small binary patterns to ensure correctness. For example, x=3 (binary 11), y=5 (binary 101) – interleaving should give: bits of x = 011, y = 101, morton = 0b011101 (which in decimal is 29). Actually step-by-step: x=011, after spreading becomes 0 1 1 with zeros in between -> 01 01 01 binary (which is 0x15), y=101 becomes (after spreading and shifted left 1) -> 10 00 10 binary (0x22), OR = 0x37 = 110111 which looks off, let's carefully do a shorter example by hand to confirm the method.

It can be tricky to validate by mental math, so writing a brute force (just loop bit by bit to interleave) to compare your function's output is helpful.

6.4 Testing for Byte Ranges in a Word

What it does: These are hacks to determine if any byte in a word meets certain criteria (is zero, equals a certain value, less than, greater than, etc.) efficiently.

Examples:

- **Has any zero byte:** We want to check if any 8-bit chunk of a 32-bit word is 0. A known trick (as we saw Day 1) was:

  c

  ```c
  bool hasZero = (((v - 0x01010101) & ~v & 0x80808080) != 0);
  ```

This sets the high bit of a byte if that byte was zero. We explained it earlier: `(v - 0x01010101)` will underflow each byte if it was 0 or make high bit 1 if originally >0x80; `~v` has high bit 1 for bytes <0x80. The combination isolates actual zeros. The result is non-zero if any byte was zero.

- **Has byte equal to n:** As shown in Day 1, XOR with `n` repeated and then use the zero-byte test: `hasvalue = hasZero(v ^ (n*0x01010101))`.
- **Has byte less than n:** Trick from the text:

  c

  ```c
  #define hasless(x, n) (((x) - ~0UL/255*(n)) & ~(x) & ~0UL/255*128)
  ```

  This formula comes from the analysis: it will yield non-zero if any byte < n. The idea: subtracting n (repeated in each byte) from x, and AND with ~x, and mask high bits. Without diving fully: it's similar to the zero-check but offset the threshold.
- **Has byte greater than n:** Provided as:

  c

  ```c
  #define hasmore(x, n) ((((x) + ~0UL/255*(127-(n))) | (x)) & ~0UL/255*128)
  ```

  This adds (127-n) to each byte and OR with original, then checks high bits.
- **Has byte between m and n:** There's a more complex formula combining the above ideas.

Understanding these thoroughly requires stepping through bit arithmetic on bytes. They essentially do parallel comparison by using constant masks like `~0UL/255` which yields a pattern like 0x01010101 (because `~0UL` is all 1s, dividing by 255 yields 0x01010101 as integer division yields quotient of 0x01010101 with remainder 0xFF). Indeed `~0UL/255 = 0x01010101`. Similarly `~0UL/255*128 = 0x80808080`.

Given the complexity, one approach to teaching this is:

- Show the formula.
- Possibly demonstrate one (like zero or equal) in detail which we did.
- For less/greater, trust the formula with a brief explanation as given in comments (the code we saw in the ghost blog lines [111-139] basically did that explanation for less and more).

Use Cases: These hacks are used for string handling (e.g. find null terminator quickly by checking in 4-byte chunks instead of byte by byte), or in vectorized algorithms where you want to check a whole word for certain byte conditions quickly. They can also count occurrences using similar logic (there were `countless` and `countmore` macros on the page as well that count how many bytes meet the condition by extending the technique).

Practice: If you're curious, implement the `hasZero` function using the given formula and test it on various words with known zero/no-zero bytes (e.g., 0x00FFFFFF has a zero in most significant byte, should return true). Similarly, try `hasvalue(x, n)` for some x and n. These are advanced and tricky – don't worry if they seem like black magic; understanding them fully might require a separate deep dive.

6.5 Lexicographically Next Bit Permutation

What it does: Given a number `v` that has N bits set (i.e., combination of N 1s in its binary form), generate the next larger number that has the same number of bits set. In other words, treat the set bits as a combination and find the next combination in lexicographic (or numeric) order.

How it works: There is a well-known algorithm for this:

1. Let `t = v | (v - 1)`. This sets all bits to the right of the rightmost 1 bit of `v`. Example: if v = 0b0110011100, then v-1 = 0b0110011011, OR gives t = 0b0110011111.
2. `w = (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(v) + 1));` – this is the compact formula given. But let's break down conceptually:
   - We need to take the pattern like c0...0100...011... to the next pattern. The typical algorithm is:
     - Determine the rightmost 1 that has a 0 to its left that can be turned to 1 (pivot).
     - Flip that 0 to 1 (pivot step).
     - Count how many 1s were to the right of that pivot originally.
     - After pivot, move all remaining 1s to the rightmost positions possible.

Another expression given is:

```c
unsigned int t = (v | (v - 1)) + 1;
unsigned int w = t | ((((t & -t) / (v & -v)) >> 1) - 1);
```

Example for clarity: For `v = 0b10011`:

- `v | (v-1) = 0b10011 | 0b10010 = 0b10011`
- `t = (v | (v-1)) + 1 = 0b10011 + 1 = 0b10100`
- `t & -t = 0b10100 & 0b01100 = 0b00100` (isolates lowest set bit of t)
- `v & -v = 0b10011 & 0b01101 = 0b00001` (isolates lowest set bit of v)
- `(t & -t) / (v & -v) = 0b00100 / 0b00001 = 0b00100 = 4` (division by 1 doesn't change value)
- `4 >> 1 = 2`
- `2 - 1 = 1 = 0b00001`
- `w = t | (1) = 0b10100 | 0b00001 = 0b10101`

So the next permutation after 0b10011 (19 in decimal, 3 bits set) is 0b10101 (21 in decimal, still 3 bits set). Let's verify manually: numbers with 3 bits between 19 and 21 are 19 (10011), 20 (10100; has 2 bits, not 3), 21 (10101; has 3 bits).

Practice: Implement this bit permutation next operation. Test cases: find the next after 0b11 (3, which has 2 bits set) – should give 0b101 (5, also 2 bits set). Also check a larger case with more bits, e.g., the next number after 0b1001101 (77, 4 bits set) which should be 0b1001110 (78, 4 bits set). Verify your algorithm by using a naive method that simply increments by 1 and checks if the bit count matches.

Day 6 Summary: Today we explored more advanced bit hacks: efficient modulo for special values (powers of 2 and powers of 2 minus 1) using bit arithmetic, interleaving bits (Morton codes) for mapping multidimensional coordinates to 1D, checking byte-level conditions across a word in parallel, and determining the next bit permutation with the same number of set bits. These represent more specialized use cases, but demonstrate the power and elegance of bit manipulation when dealing with certain problems. Tomorrow, we'll wrap up with a review and additional byte-level tricks.

Day 7: Byte-Level Tricks and Next-Bit Permutation

Topics: Advanced byte-wise tests within a word (checking if any byte is zero, equal to a value, less/greater than a value), and generating the next combination of a fixed number of 1-bits (lexicographically next bit permutation). These are more specialized hacks that showcase clever use of arithmetic to manipulate bytes in parallel and to handle combinatorial bit patterns.

7.1 Byte-Wise Tests in a Word (Zero, Equal, Less, Greater, Between)

These hacks treat a 32-bit (or 64-bit) word as a collection of bytes and perform comparisons on all bytes simultaneously using bit arithmetic. The key idea is to construct a mask that flags bytes meeting the condition.

- **Has Any Zero Byte:** A classic trick to detect if any byte in a 32-bit integer is 0 uses the expression:

  c

  ```c
  bool hasZero = (((v - 0x01010101) & ~v & 0x80808080) != 0);
  ```

  How it works: `0x01010101` is a value with 0x01 in each byte. Subtracting that from `v` will make any byte that was 0 underflow to 0xFF (setting that byte's high bit to 1), or if a byte was >=0x01, the high bit might also set if it was originally >0x80. Meanwhile, `~v` has 1s in positions where `v` had 0s (so if a byte of `v` was 0x00, that byte in `~v` is 0xFF, which has high bit 1; if `v`'s byte was non-zero, ~v's byte has high bit 0 only if original was <0x80). By ANDing these: `(v - 0x01010101)` (which has high bits set for bytes that were 0 or >0x80) with `~v` (high bits 1 for bytes that were <0x80) and `0x80808080` (to isolate high bits of each byte), we get non-zero if and only if a byte was originally zero. In summary, this sets a flag in the high bit of each byte that is zero, and none of the others, so the check `!=0` detects presence of any zero byte.

- **Has Byte Equal to n:** We can reuse the zero-byte check by first XORing each byte with the target value *n*. If a byte equals `n`, then after XOR with `n` it becomes 0x00. So:

  ```c
  bool hasValue = (((v ^ (n * 0x01010101)) - 0x01010101) & ~(v ^ (n * 0x01010101)) &
  ```

  In macro form from the Bit Hacks page:

  ```c
  #define hasvalue(x, n) (haszero((x) ^ (~0UL/255 * (n))))
  ```

  Here `~0UL/255` computes 0x01010101 (since `~0UL` is all 1s and dividing by 255 yields 0x01010101), so `~0UL/255 * n` repeats the byte `n` across the word (e.g., for n=0x7F, this is 0x7F7F7F7F). XORing `v` with that will produce 0x00 in any byte that was equal to `n`, and something non-zero otherwise. Then it applies the zero-byte detection on the result. If any byte was equal to `n`, the expression is non-zero.

- **Has Byte Less Than n:** This one is more involved. One provided formula is:

  ```c
  #define hasless(x, n) ((((x) - ~0UL/255 * (n)) & ~(x) & ~0UL/255 * 128) != 0)
  ```

  In essence, `(x - 0x01010101 * n)` will underflow in any byte where `x`'s byte < `n`, causing that byte's high bit to become 1 (because of two's complement wraparound). It will also produce some high bits = 1 for bytes where `x`'s byte was ≥ `n` + 128 (due to overflow of subtraction), but the next part `& ~(x)` zeros out high bits for bytes where `x`'s original byte had its high bit 1 (i.e., originally ≥128, to filter out false positives). Finally, `& ~0UL/255 * 128` masks to only the high bit of each byte (0x80 in each byte position). The result will have 0x80 in a byte if and only if that original byte was < n. Thus the check `!=0` tells us if any byte was less than the value `n`.

- **Has Byte Greater Than n:** Similarly, a macro from the page:

  ```c
  #define hasmore(x, n) ((((x) + ~0UL/255 * (127 - (n))) | (x)) & ~0UL/255 * 128)
  ```

  This one adds `(127-n)` to each byte of `x`. If a byte was greater than `n`, adding `(127-n)` will carry out of that byte (because byte > n implies byte + (127-n) > 127, which overflows the 7-bit range and sets the 8th bit). The `| (x)` then ensures the high bit is also set for bytes that were already ≥128 (since those are trivially "> n" if n ≤ 127). The mask with `0x80808080` again isolates the high bits. The result non-zero means some byte > n. (This one is quite subtle: it effectively checks if byte > n in two parts: bytes 1–127 handled by the addition overflow, bytes ≥128 always qualify as > n and are caught by OR with original x.)

- **Has Byte Between m and n (exclusive):** An even more complex combination exists for testing a range m < byte < n. The page gives macros `likelyhasbetween` and `hasbetween` which use a combination of the techniques above (essentially checking `hasless(x, n)` and `hasmore(x, m)` simultaneously with some adjustments for exclusive range). We won't detail it here due to complexity, but it follows from combining the "less" and "greater" logic.

These byte-wise hacks are quite advanced and it's normal if the logic isn't immediately obvious. They cleverly use arithmetic that carries into the high bit (overflow/underflow) to act as the result of a comparison, then isolate that with masks.

Use Cases: Fast parsing of text/bytes: e.g., checking if a 32-bit chunk of a string has a null terminator (zero) – this trick is used in optimized C string functions to check 4 bytes at a time. Similarly, checking for characters in a range (like alphanumeric checks) could use these. It's also useful in data processing where you want to quickly find bytes that meet criteria without branching per byte.

Practice: To solidify understanding, you might pick one of these (say, `hasZero` and `hasByteEqual`) and test them in a program. For `hasZero`, try values like `0x00FFFFFF` (should be true, as high byte is 0), `0xFFFFFFFF` (false), `0xFF00FF00` (true, some zero bytes). For `hasByteEqual(x, 0x20)` (to check if any byte is space character), test it on packed 4-character chunks. Even if the formulas are complex, testing will at least convince you they work.

7.2 Lexicographically Next Bit Permutation

Imagine you have an integer `v` with a fixed number of 1-bits (say N bits set to 1). You want to get the next higher integer that has the same number of 1-bits (the next combination in sorted order). For example, if N=3 and `v = 0b00010011` (which is 0x13, bits at positions 0,1,4 set), the next combinations in increasing order would be: `0b00010101` (positions 0,2,4), `0b00010110` (positions 1,2,4), `0b00011001`, ... up to `0b111000` for N=3. We want a formula or method to go from one combination to the next.

How it works: A known algorithm is:

1. Identify the **rightmost 1-bit** that can be "moved" to a higher position. This is the pivot point where a `01` sequence (0 then 1) will become `10`. In binary, find the pattern `...01...` where to the right of that pattern, all bits are 1s (because those are the ones that will "slide" to the lowest positions).
2. Flip that `01` to `10` (meaning: change that 0 to 1 and that 1 to 0 – effectively incrementing that 1-bit's position).
3. After the pivot, you will have some number of 1-bits (say `r` ones) that were to the right. To get the next smallest number, you should put those `r` ones in the **rightmost** positions possible (i.e., make the number as small as possible while still larger than the original). That typically means after the pivot, put all the remaining 1s in the least significant bit positions.

In practice, a common implementation is:

- Let `c = v & -v` (this isolates the lowest set bit of v).
- Let `r = v + c` (adds that bit to v, effectively flipping the lowest `01` pattern to `10`...).
- Compute `w = r | ((((r & -r) / c) >> 1) - 1)`. This cryptic formula does the redistribution of the trailing ones.
  - `(r & -r)` gives the new lowest set bit in the result (which corresponds to the bit that was 0 and became 1 at the pivot).
  - Divide that by `c` (the old lowest set bit) gives a value that is a power of two plus some - it effectively measures how far we shifted.
  - `(((r & -r) / c) >> 1) - 1` yields a number with a certain number of low bits set to 1 – specifically `r` has one more 1 than v did at a higher position, and this expression creates a mask of `r` trailing 1s to place at the bottom.
  - ORing that with `r` gives the final next permutation.

The Bit Twiddling Hacks page provides two equivalent formulas:

```c
unsigned int t = v | (v - 1);
// t gets v's bits to the right of the least significant 0 (inclusive) all set to 1.
unsigned int w = (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(v) + 1));
```

and an alternate (which avoids the need for `ctz` (count trailing zeros)):

```c
unsigned int t = (v | (v - 1)) + 1;
unsigned int w = t | ((((t & -t) / (v & -v)) >> 1) - 1);
```

Either will compute the next number `w` with the same number of 1 bits.

Example: Say `v = 0b00010011` (with N=3 ones). Following the algorithm:

- c = 0b00000001 (lowest set bit).
- r = v + c = 0b00010100 (this basically added 1 to the part of v beyond the lowest set bit, effectively pivoting the 0 at bit 2 and the 1 at bit 1 to 1 at bit 2 and 0 at bit 1).
- Now, `v` had two 1s (at bit0 and bit1) below that pivot. After r, those two 1s are gone (everything right of bit2 cleared). We need to put those 2 ones back into the lowest positions. The formula `(((r & -r) / c) >> 1) - 1` will compute a mask of 2 ones. Indeed,
  - r & -r = 0b00000100 (isolates the new lowest set bit in r, which is bit2).

    - Divide by c (0b1) gives 0b00000100. >>1 gives 0b00000010. Subtract 1 gives 0b00000001 – this is actually 1 one, not 2... I need to carefully apply it: Actually, let's use the second formula for clarity:
    - t = (v | (v-1)) + 1. For v=00010011, v| (v-1) = 00010011 | 00010010 = 00010011 (since v-1 only clears lowest set bit and sets lower bits). +1 gives t = 00010100 (same as r).
    - Now `(t & -t)` = 0b00000100, `(v & -v)` = 0b00000001, divide: 0b100 / 0b1 = 0b100, >>1 = 0b010, minus 1 = 0b001. OR with t: 00010100 | 00000001 = 00010101, which is the next permutation (0x15) as expected.
- Indeed, 0b00010101 is the next combination (bits at positions 0,2,4).

Use Cases: Generating combinations in algorithms, e.g., iterating through subsets of a fixed size represented as bits. Also useful in some brute-force or search algorithms where you need to enumerate bit patterns systematically.

Practice: Implement a function `next_combination(x)` that returns the next bit permutation after `x` (with the same count of 1s). Use one of the above formulas or the step-by-step approach: find pivot, construct result. Test it on sequences of combinations to ensure it generates correctly. Start with an initial pattern like `0b00111` (3 ones in the lowest positions) and repeatedly call `next_combination` – it should enumerate all 3-one combinations up to `0b11100`.

Day 7 Summary: We delved into two advanced topics: **byte-level parallel comparisons**, which show how a single arithmetic expression can simultaneously evaluate a condition on all bytes of a word (an impressive feat that leverages overflow behavior), and **next bit permutation**, a clever algorithm to treat bit patterns as combinations and find the next one in order. These hacks are on the more complex end of the spectrum, and it's okay if it takes some effort to wrap your head around them. When implementing such hacks, careful testing is essential.

Conclusion and Final Tips

Over this week, we've explored a wide array of bit twiddling hacks — from simple tricks to eliminate branches, to advanced manipulations and binary algorithms. To conclude, here are a few general tips when working with these techniques:

- **Visualize and Verify:** Draw out binary representations or use small examples to trace through how a hack works. For instance, write down an 8-bit version of the problem and manually apply the operations to see the result. This greatly aids understanding and debugging.
- **Use Built-ins When Available:** Modern languages/compilers often have intrinsics for operations like counting bits (`__builtin_popcount`), counting trailing zeros (`__builtin_ctz`), etc. These are usually implemented using optimal CPU instructions. If performance is critical and such intrinsics are available, use them – they embody these hacks under the hood.

- **Beware of Undefined Behavior:** Some tricks rely on specific behavior (e.g., right shifting signed integers, overflow in C which is undefined for signed types, etc.). When implementing, prefer unsigned types for bit tricks and know your language's rules. The hacks we presented often use unsigned arithmetic to avoid undefined behavior.
- **Edge Cases:** Pay attention to extreme values (all bits 0 or all bits 1, etc.). For example, the "next permutation" trick won't work if `v` is already the last permutation (e.g., 0x111000... if all 1s are already in the top positions), so handle that if needed. Similarly, some algorithms needed a special-case for `v=0`.
- **Comment Your Code:** While you might now understand the hack, it can look like gibberish to someone else (or to yourself, a year later!). A brief comment, like "// isolate lowest set bit" or "// branchless abs using mask", helps maintain readability.
- **Measure vs. Simpler Code:** Not all hacks are actually faster on modern hardware – compilers are smart, and sometimes a straightforward approach is just as good. Use these hacks when they truly benefit (e.g., in tight loops or where branch misprediction is a problem). Always profile if performance is critical.

By now, you should have a solid toolkit of bit tricks. You've seen how powerful bit manipulation can be in optimizing low-level code and solving problems elegantly. With practice, identifying when and how to apply these hacks will become more intuitive. Happy bit twiddling!b10100110; // binary for illustration

```
int i = 1, j = 5;          // swap bit1 and bit5 (0-indexed from LSB)
unsigned int bit1 = (v >> i) & 1;
unsigned int bit5 = (v >> j) & 1;
if (bit1 != bit5) {
v ^= ((1U << i) | (1U << j));
}
printf("New v = 0b%08b\n", v); // use %08b assuming a custom binary print function
```

~~~
Before swap: `v = 10100110` (bit5=1, bit1=1). After swap, since bit1 and bit5 were
both 1 (the same), it should remain unchanged (and indeed the `if` will skip
flipping). If you choose bits that differ, e.g. swap bit1 and bit2 in the original
(`bit1=1, bit2=0`), the mask will flip them and you'll see the result with those
bits exchanged.

If you want to swap bits in two different variables (say bit i of x with bit j of
y), you can similarly do:
```c
unsigned int bitMaskX = (1U << i);
unsigned int bitMaskY = (1U << j);
unsigned int xi = (x >> i) & 1;
unsigned int yj = (y >> j) & 1;
if(xi != yj) {
    x ^= bitMaskX;
    y ^= bitMaskY;
}
~~~

This flips the differing bits in their respective numbers.

Use Cases: Bit-level permutation is useful in low-level data processing, cryptography (swapping bits in certain algorithms or bit-scrambling), and sometimes in hardware simulation.

Practice: Write a function to swap any two bits of an integer using the above logic. Test it by printing the binary representation before and after to confirm the swap. Try edge cases like swapping a bit with itself (which should do nothing) or swapping bits when both are 0 or both are 1 (which should also result in no change to the number's value). Ensure your code respects those conditions.

4.4 Reversing Bits – Simple Loop Method

What it does: Reverses the order of bits in a word. For example, if you have an 8-bit value `v = ABCDEFGH` (bits), the reverse is `HGFEDCBA`. We want to do this for, say, a 32-bit integer (producing a reversed 32-bit output).

How it works: The straightforward method is to use a loop that takes bits from one end of the number and builds the reversed result:

- Initialize `rev = 0`.
- For bit position i from 0 to N-1 (N= word length, e.g., 31):
  - `rev <<= 1` (make space for a new bit on the right).
  - `rev |= (v >> i) & 1` (take the i-th bit of `v` and put it as the new LSB of `rev`). This effectively shifts `rev` left and appends the next bit of `v`. By the end, `rev` holds the reversed

bit sequence.

Example – C Implementation (reverse bits loop):

c

```c
unsigned int v = 0
```