## Some useful bitwise algorithms/bit hacks

1. **Checking if a number is a power of 2**
   **Algorithm:**

   The numbers which are powers of 2 have only one set bit in their binary representation. If the number is 0 or is not a power of 2 then it will have more than one bit set in binary representation.
   Let us say, we need to check if a number x is a power of 2 or not.
   Now, think about the binary representation of (x-1), the binary representation of (x-1) will have all the bits the same as x except for the rightmost set bit in x and all the bits to the right of the rightmost set bit.

   **For example:**
   $x = 4\ (00100)_2$
   $x-1 = 3\ (00011)_2$
   $x = 10\ (1010)_2$
   $x-1 = 9\ (1001)_2$

   Now, how (x-1) can help determine whether x is a power of 2?
   The binary representation of (x-1) can be simply obtained from the binary representation of x by flipping all the bits to the right of the rightmost set bit including the rightmost set bit itself, so if x has only one-bit set, then x & (x-1) must be equal to 0.

   **For example:**
   $x = 4\ (00100)_2$
   $x-1 = 3\ (00011)_2$
   $x\ \&\ (x-1) = (00000)_2$

   In the above example, since the rightmost set bit and all the bits to the right of it were flipped, so the bitwise and will make all those bits 0 and we have no set bits left, which tells us that there was only one set bit in x.

   **Pseudocode:**

   ```
   function isPowerOf2(num)
           return (num && ! (num & (num - 1)))
   ```

   A similar approach can be used for checking if a given number is **even** or **odd.**
   One of the obvious approaches would be to check the number's divisibility by 2, but the other approach is to check if the last(rightmost) bit i.e if the last bit is 1 or not if the last bit is 1 then the number is odd, otherwise even, as the rightmost bit i.e 2^0 only contributes odd value - 1 to the number, all other bits represent the value of 2^i where i > 0, so all values remain even.

2. **Checking if the kth bit is set/unset**
   **Algorithm:**

   We need to check if the kth bit in the binary representation of the given number is set or unset. We can use bitwise AND operations to check this efficiently.
   We can take another number n = (1<<k), whose all bits are 0 except the kth bit.
   Now doing AND operation with the given number say x, must return a positive number if the kth bit is set, otherwise the result of bitwise AND will be 0.

   **For example:**
   x = 5 $(101)_2$ , k = 2
   So, let no = (1 << k) = 2^k = 4 = $(100)_2$
   x & no = $(101)_2$ & $(100)_2$ = $(100)_2$ = 4.
   So the $2^{nd}$ bit of x is set.

   **Pseudocode:**

```
function checkBit(N, k)
        if (N & (1<<k))
                return true
        else
                return false
```

3. **Counting number of 1s in the binary representation of a number**
   The naive approach is to simply convert the given number into its binary representation and simply count the number of 1s in the binary representation, which takes $O(\log_2 N)$ time, where N is the number.

   However, we can efficiently count the number of 1s in the binary representation of a number whose time complexity depends on the number of 1s in its binary representation.

   **Algorithm:**
   Let us say we want to find the number of 1s in the binary representation of x, we can perform a similar operation used to check if a given number is a power of 2.
   We can apply bitwise and between x and (x-1) and then update the value of x to x & (x-1), every time this operation flips the rightmost set bit and the bits after the rightmost set bit, so every time the rightmost set bit is unset, which helps us count the number of 1s in the binary representation of the number.
   **For example:**
   x = 5 $(101)_2$
   1. x = 5 $(101)_2$

x-1 = 4 $(100)_2$

x & (x-1) = $(100)_2$ = 4, update x to 4.

2. x = 4 $(100)_2$

   x-1 = 3 $(011)_2$,

   x & (x-1) = $(000)_2$ = 0

So, the number of set bits are 2 in x.

The above algorithm described is known as **Brian Kernighan's Algorithm.** The time complexity of the algorithm is dependent on the number of 1s in the binary representation of the number, however in the worst case still the time complexity will be $O(\log_2 N)$ as all bits of the number may be set, where N is the number.

**Pseudocode:**

```
function count1s(num)
        count = 0
        while(num > 0)
                num = num & (num-1)
                count = count+1
        return count
```

4. **Generating all possible subsets of a set**

Bit manipulation can be useful for the generation of all the possible subsets of a set. We can represent the elements of the set in the form of a binary sequence having a length equal to the number of elements in the set. Since the number of subsets of a set having N elements is 2^N, so a binary sequence of length N will represent values from range 0 to 2^N-1 which is equal to the number of subsets of the set.

All set bits in the binary sequence denote the elements present from the set, thus representing a subset.

**For example:**

Set S = {10,20,30}

We know that the number of subsets of a set having N elements is 2^N.In this case N = 3, so 2^N = 8 subsets.

So taking a binary sequence of length 3, where each bit value represents whether the element of the set is present(bit value is 1) or not(bit value is 0) in the corresponding subset.

0 = 000 => {} - empty subset

1 = 001 => {30}

2 = 010 => {20}

3 = 011 => {20,30}

4 = 100 => {10}

5 = 101 => {10,30}

6 = 110 => {10,20}
7 = 111 => {10,20,30}

Hence, the binary sequence is useful to represent all the subsets of a set.

**Pseudocode:**

```
function generateSubsets(arr, N)
        /*
                arr represents the set of size N
                Iterating from 0 to 2^N-1 to get all subsets
        */
        for i = 0 to 2^N-1
                /*
                        For each i representing a subset checking which
                        bits are set in binary representation of i
                */
                for j = 0 to N-1
                        if (i & (1 << j))
                                print arr[j]
                print newline
        return
```

5. **Some other bitwise tricks**
   - **res = x | (1 << k):** Set the bit at kth position in the binary representation of x.
     **For example:**
     x = 5 $(101)_2$ , k = 1
     Let no = (1 << k) = 2^1 = 2 = $(010)_2$
     res = x | no = $(101)_2$ | $(010)_2$ = $(111)_2$ = 7
   - **res = x & (~(1 << k)):** Unset the bit at kth position in binary representation of x.
     **For example:**
     x = 5 $(101)_2$ , k = 2
     Let no = (1 << k) = 2^2 = 4 = $(100)_2$
     no = ~(no) = $(011)_2$
     res = x & no = $(101)_2$ & $(011)_2$ = $(001)_2$ = 1
   - **res = x ^ (1 << k):** Toggle the bit at kth position in the binary representation of x i.e if the bit at kth position is 0 then change it to 1 and vice versa.
     **For example:**
     x = 5 $(101)_2$ , k = 0
     Let no = (1 << k) = 2^0 = 1 = $(001)_2$
     res = x ^ no = $(101)_2$ ^ $(001)_2$ = $(100)_2$ = 4
   - **res = x & (x - 1):** Unsetting the rightmost set bit of x.
     **For example:**

$x = 5\ (101)_2$
$x-1 = 4\ (100)_2$
$res = x\ \&\ (x-1) = (101)_2\ \&\ (100)_2 = (100)_2 = 4$

- **res = ~x + 1 (2's complement):** 2's complement of a number is its 1's complement +1. 2's complement of a number is the same as negative of the number i.e -x.
- **res = x & (-x):** Getting the rightmost set bit of x.
  **For example:**
  $x = 5\ (101)_2$
  1's complement: $\sim x = (010)_2$
  2's complement: $\sim x + 1 = (011)_2$
  $res = x\ \&\ (-x) = (101)_2\ \&\ (011)_2 = (001)_2 = 1$

## Applications of bit manipulation

1. Bitwise operations are prominent in embedded systems, control systems, etc where memory(data transmission/data points) is still an issue.
2. They are also useful in networking where it is important to reduce the amount of data, so booleans are packed together. Packing them together and taking them apart use bitwise operations and shift instructions.
3. Bitwise operations are also heavily used in the compression and encryption of data.
4. Useful in graphics programming, older GUIs are heavily dependent on bitwise operations like XOR(^) for selection highlighting and other overlays.