Contents

1 Bit Manipulation

Reference

1.1 Introduction

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Computer programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization. For most other tasks, modern programming languages allow the programmer to work directly with abstractions instead of bits that represent those abstractions. Source code that does bit manipulation makes use of the bitwise operations: AND, OR, XOR, NOT, and bit shifts.

Bit manipulation, in some cases, can obviate or reduce the need to loop over a data structure and can give many-fold speed ups, as bit manipulations are processed in parallel, but the code can become more difficult to write and maintain.

1.2 Basics

At the heart of bit manipulation are the bit-wise operators:

- `&`: AND
- `|`: OR

- `~`: NOT
- `^`: XOR (exclusive OR), equivalent to adding two bits and discarding the carry. XOR is its own inverse, this means for a given number `num`, `num ^ num == 0`. Also, `num ^ 0` is `num` itself.
- `«`: left shift
- `»`: right shift

There is no boolean operator counterpart to bitwise exclusive-OR, but there is a simple explanation. The exclusive-OR operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-OR, with the operator of a caret, `^`, performs the exclusive-OR operation on each pair of bits. Exclusive-OR is commonly abbreviated XOR.

Some other manipulations are (`A` and `B` are two integers):

- set union: `A | B`
- set intersection: `A & B`
- set substraction: `A & ~B`
- set negation:
  - `ALL_ONE_BITS ^ A`
  - `~A`: pay attention that this is not A -> -A, to make A -> -A, you do: `~(A - 1)`.
- set bit: `A |= 1 « bit`: set the bit with a position of `bit` as 1 (from right to left)
- clear bit: `A &= ~(1 « bit)`: set the bit with a position of `bit` as 0 (from right to left)
- test bit: `(A & 1 « bit) != 0`: test the bit with a position of `bit` to see if it is 1 or 0
- extract last bit: get the significance of the last 1-bit (its position), keeping all zeros from the last bit to rightmost zero.
  - `A & -A`
  - `A & ~(A - 1)`
  - `A ^ (A & (A - 1))`: you first remove the last bit, then XOR with the original number, you'll only keep the last bit since it is the only place where two numbers different.
- remove last bit: `A & ( A - 1)`
- get `ALL_ONE_BITS`: `~0`

1.3 Examples

1.3.1 Count the number of one bit (practice `&`, `«`)

Count the number of ones in the binary representation of the given number. The strategy is to count the rightmost one in its binary representation and remove it, until you remove all the ones in its binary representation.

To remove the rightmost one in a number's binary representation, you do: `n & (n - 1)`.

Take a closer look on `(n - 1)`. What it does is actually transform all zeros from rightmost one to the right end to 1, and transform the rightmost one to zero. See below examples:

```
eg. 1.
1024: 0100 0000 0000
1023: 0011 1111 1111

eg. 2.
44: 0010 1100
43: 0010 1011
```

Pay attention that from rightmost one to right end, all digits are 0 (apparently this is how we find rightmost digit 1).

Then, if we do `n & (n - 1)`, the effect is like we remove the rightmost digit 1:

```
eg. 1.
    1024: 0100 0000 0000
    1023: 0011 1111 1111
1024 AND 1023:
    xxxx: 0000 0000 0000
    1024: 0100 0000 0000
compared with 1024, we removed rightmost digit 1

eg. 2.
    44: 0010 1100
    43: 0010 1011
44 AND 43:
    xx: 0010 1000
    44: 0010 1100
compared with 44, we removed rightmost digit 1
```

Once we know how to remove the rightmost digit 1 (`n & (n - 1)`), the method to count the number of ones is straightforward:

```
int countOne(int n) {
  int count = 0;

  while (n) {
```

```
    n = n & (n - 1);
    count++;
  }

  return count;
}
```

1.3.2 Is power of four (practice &)

If a number is a power of four, it has only one digit 1, and this single 1 should be in any of the following positions occupied by one:

```
0101 0101 0101 0101 0101 0101 0101 0101

example, all following numbers are power of four:
0000 0000 0000 0000 0000 0000 0000 0001 (1)
0000 0000 0000 0000 0000 0000 0000 0100 (4)
0000 0000 0000 0000 0000 0000 0001 0000 (16)
0000 0000 0000 0000 0000 0001 0000 0000 (256)
```

The number in hexadecimal form is: `0x55555555`.

For a given number `n`, we have to check if it has only one 1 in its binary form (otherwise, it cannot be the power of 4). Using knowledge from previous section, this can be simply `!(n & (n - 1))`. The idea is straightforward, if we remove the rightmost digit 1 from the number, and the number becomes zero, it means there is only a single digit one in that number.

Then, we try to find out the location of this 1-bit.We do `n & 0x55555555`. For example, take `n = 512`:

```
    0101 0101 0101 0101 0101 0101 0101 0101
AND 0000 0000 0000 0000 0000 0010 0000 0000
    0000 0000 0000 0000 0000 0000 0000 0000
```

The 1-bit in 512 is not coincide with any 1-bit in `0x55555555`, so 512 is not a power of 4. The result of `AND` is zero (boolean value false).

Let's see `n = 4194304`:

```
    0101 0101 0101 0101 0101 0101 0101 0101
AND 0000 0000 0100 0000 0000 0000 0000 0000
    0000 0000 0100 0000 0000 0000 0000 0000
```

This time, the 1-bit in 4194304 coincides with one 1-bit in `0x55555555`, so it is a power of 4. The result of `AND` is one (boolean value true).

Wrap up the solution, we have:

```
int isPowerOfFour(int n) {
  return !(n & (n - 1)) && (n & 0x55555555);
```

```
}
```

1.3.3 Add two numbers (practice `^`, `&`, `«`)

Add two numbers without using `+`, `-`, `++`, `--`.

Recursion (This method only works for two positive number). –update: this method also works for negative numbers. Just declare the two integers as `unsigned int`. Otherwise, leetcode will complain about left shift of a negative number

The function prototype is:

```
int getSum(int a, int b);
```

Remember the exclusive OR (`^`) is equivalent to adding two bits and discard carry. Also, adding two numbers is essentially, adding each pairing bits. It may have carry or not. The situations are listed below (for a single bit addition):

```
No carry:
1    0    0
0    1    0
- or - or -
1    1    0

With carry:
 1
 1
 -
10
```

So, we can split the sum in two parts. Part 1 comes from adding two numbers without carry, which is simply `a ^ b`. Part 2 comes from the possible carry from the addition. We can get this value by first `a & b` (this is give us 1-bit in both `a` and `b`), then we left shift this number (because carry is essentially shifting 1 to next left bit). So the carry part is `(a & b) « 1`. Now we have thise two parts, we can call `getSum()` again to calculate the sum for us.

The base case is when the carry part is zero. In this case we can simpliy return the other part. The code is as follows:

```
int getSum(int a, int b) {
  if (b == 0)  // when the carry part is zero, a ^ b is a + b, so return
  ↪  the first part directly
    return a;

  return getSum(a ^ b, (a & b) << 1);
}
```

Add directly Reference This approach adds each bit in number `a` and number `b` to get the final result. To achieve this, we need following variables:

- `loop`: an integer that record the position of current adding bit.
- `sum`: an integer, keep the running total of summation
- `ai`: an integer, keep the bit value of `a` at position specified by loop
- `bi`: an integer, keep the bit value of `b` at position specified by loop
- `ci`: an integer, keep the bit value of `sum` at position specified by loop

When traversing the number, we extract each bit-value of `a` and `b` and add them to `sum` (including carry, if there is any). We use following sequences to loop through all bits in an integer:

```
0000 0000 0000 0000 0000 0000 0000 0001
0000 0000 0000 0000 0000 0000 0000 0010
0000 0000 0000 0000 0000 0000 0000 0100
0000 0000 0000 0000 0000 0000 0000 1000
...
0010 0000 0000 0000 0000 0000 0000 0000
0100 0000 0000 0000 0000 0000 0000 0000
1000 0000 0000 0000 0000 0000 0000 0000
```

We can get this use a while loop and left shift:

```
int loop = 1;
while (loop) {
  // ...
  loop = loop << 1;
  // ...
}
```

To extract the bit value where 1-bit in loop at, we simpliy:

```
int loop = 1;
int sum = 0;
while (loop) {
  ai = a & loop;
  bi = b & loop;
  ci = sum & loop;

  loop = loop << 1;
}
```

Then, we first add the bit value in `ai` and `bi` to the same position in `sum`, discarding carry right now:

```
int loop = 1;
int sum = 0;
while (loop) {
  ai = a & loop;
  bi = b & loop;
  ci = sum & loop;
  sum = sum ^ ai ^ bi;
  loop = loop << 1;
}
```

Now we add carry. A carry should be added as long as any two of `ai, bi, sum` has a bit value of one. Also, only one carry is possible (you can draw a table listing all possible values of `ai, bi, sum` to verify this). Thus, the condition to add a carry is:

```
(ai & bi) || (ci & ai) || (ci & bi)
```

To add a carry, we first move `loop` one bit left, then we do `sum ^ loop`. This is because the bit at `loop`'s current position (incremented) is 1 and the bit at `sum` is still 0, also, all other bits in `sum` will be kept (because all other bits in `loop` is 0). So we utilize the exclusive OR again:

```
int loop = 1;
int sum = 0;
while (loop) {

  ai = a & loop;
  bi = b & loop;
  ci = sum & loop;

  sum = sum ^ ai ^ bi;
  loop = loop << 1;

  if ((ai & bi) || (ci & ai) || (ci & bi))
    sum = sum ^ loop;
}
```

Another thing worth noticing is, when `loop` reaches:

```
1000 0000 0000 0000 0000 0000 0000 0000
```

If we left shift it one more time, it becomes:

```
0000 0000 0000 0000 0000 0000 0000 0000
```

which is zero, then the while loop terminates. After the while loop terminates, we then return whatever inside `sum` as the summation result.

Interestingly, this method also works for adding numbers with different sign (negative numbers). But I don't know exactly why. I traced some examples, if the abs(negative) >

abs(positive), then the leading 11111s (two-complements for negative number) will not disappear, following structure will kind of "regenerate" them:

```
0  this is 0 from positive operand
1  this is 1 from negative operand
0  this is carry (no carry)
-
1  results after adding carry, which is 1
```

the final result is negative.

if abs(negative) < abs(positive), all the leading 11111s will get eradicated one by one:

```
0  this is 0 from positive operand
1  this is 1 from negative operand
1  this is carry
-
0  results after adding carry, which is 0, carry goes to next bit
```

the final result is positive.

Code:

```
int getSum(int a, int b) {
  unsigned int ai;
  unsigned int bi;
  unsigned int ci;

  unsigned int sum = 0;
  unsigned int loop = 1;

  while (loop) {
    // extract current bit of a, b and sum
    ai = a & loop;
    bi = b & loop;
    ci = sum & loop;

    // add to sum, discard carry
    sum = sum ^ ai ^ bi;

    // move position
    loop = loop << 1;

    // add carry if there is any
    if ((ai & bi) || (ci & ai) || (ci & bi))
      sum = sum ^ loop;
  }
```

```
  return sum;
}
```

1.3.4 Missing number (practice ^)

Given an array containing n distinct numbers taken from 0, 1, . . . , n, find the one that is missing from the array.

We'll use two properties of `^`:

1. the XOR is its own inverse, which means `num ^ num == 0`
2. any number XOR with zero is the number itself

From the problem we know that the range of index would be `0` to `n - 1`, and possible value range is `0` to `n`. Assume the array is sorted in ascending order (for illustration purpose, it still works for un-sorted array). We have two cases to consider: (1) the missing number is less than `n`; (2) the missing number is `n`;

Case 1: missing number is less than n Assume the missing number in array is `k`. We traverse the array from index `0` to `n - 1`. We use `0` to XOR all index and all `nums[index]`, we'll get:

```
0^0^0^1^1^...^(k-1)^(k-1)^k^(k+1)^...^(n-1)^n
```

add parenthese to make it clearer:

```
0 ^ (0^0) ^ (1^1) ^...^ ((k-1)^(k-1)) ^ (k^(k+1)) ^...^ ((n-1)^n)
```

The first zero is what we use to XOR all index and `nums[index]`. The first term in parenthese is index, which ranges from `0` to `n - 1`. The second term is `nums[index]`. We XOR them one by one. Notice that when index is `k`, `nums[k]` is `k + 1`, because `k` is missing in `nums`. It is clear that if the array is unsorted, we can still get this structure by rearranging terms, since XOR is associative. Now, rearranging same terms together:

```
0 ^ (0^0) ^ (1^1) ^...^ ((k-1)^(k-1)) ^ ((k+1)^(k+1)) ^...^ ((n-1)^(n-1)) ^ n ^ k
```

`k` is put to the end.

Now, use `^`'s self-inverse property, we cancel out `(i^i)` terms:

```
0 ^ n ^ k
```

to find out the missing `k`, we just need to XOR this with the size `n`:

```
0 ^ n ^ n ^ k = 0 ^ k = k
```

Case 2: missing number is n Using same technique, we use `0` to XOR all index and all `nums[index]`, we'll get:

```
0^0^0^1^1^...^(n-1)^(n-1)
```

Cancel out all `(i^i)` terms, we have `0` left. Then if we XOR it with the size `n`, we still get the missing number (in this case, `n`). So, for both cases, we can use the same technique to solve it.

Solution Get the result of `0` XOR each index (from `0` to `n - 1`) and each number (`nums[i]`), then XOR the result with the size of the array `nums.size()`.

The code is as follows:

```
int missingNumber(vector<int>& nums) {
  int ret = nums.size();

  for (int i = 0; i < nums.size(); i++) {
    ret ^= i;
    ret ^= nums[i];
  }

  return ret;
}
```

1.3.5 Largest power of 2 (practice |)

Given a number `num`, find the largest power of 2 (the most significant bit in binary form) which is less than or equal to `num`.

Solution 1: remove right 1-bit until only one left This method uses the same technique as counting 1-bit. We remove the rightmost 1-bit until only a single 1-bit left:

```
long largest_power(long N) {
  int temp;

  while (temp = N & (N - 1))
    N = temp;

  return N;
}
```

Solution 2: use | The `|` operator will keep as many 1-bits as possible. Here, the thought is first changing all right side bits to 1, then we add 1 to the number and perform a right shift.

```
long largest_power(long N) {
    //changing all right side bits to 1.
    N = N | (N>>1);
    N = N | (N>>2);
    N = N | (N>>4);
```

```
    N = N | (N>>8);
    N = N | (N>>16);
    return (N+1)>>1;
}
```

1.3.6 Reverse bits (practice |)

Given an unsigned integer (32 bits), reverse the order of the bits.

Extract each bit and store the bit in another number We traverse the number, extract the value on each bit in order, then we store this bit-information in another number (for example, we can use a zero integer to hold the result) in reverse order.

To traverse the whole number, we use a loop number, or mask, which is 1 initially, and left shift one bit at each iteration.

The code is as follows:

```
uint32_t reverseBits(uint32_t n) {
  uint32_t ret = 0;  // to hold result
  uint32_t pos = 1;  // to loop over the num, mask

  for (int i = 31; i >= 0; i--, pos <<= 1)
    ret |= ((n & pos) == 0) ? 0 : 1 << i;  // (n & pos) should be in
    ↪ parenthese

  return ret;
}
```

`(n & pos)` should be in parenthese, otherwise `pos == 0` will be evaluated first, then the truth value will be used to calculate `n & bool`.

Solution 2: right shift input number repeatedly and collect rightmost bit The rightmost bit in `n` can be extracted easily by `n & 1`. To get the rightmost bit of `n` one by one, we can simpliy right shift it after we get the current rightmost bit. We store the information of extraced bit value in reverse order, i.e. from leftmost bit to rightmost bit, to a container initially set as zero (`ret`). Thus, the mask will be `1 « 31` this time (because the mask is used to record bit, not to extract value, when the current rightmost bit in `n` is 1, we do `ret |= mask` to record this 1-bit in the corresponding position in `ret`).

The code is as follows:

```
uint32_t reverseBits(unit32_t n) {
  unsigned int mask = 1 << 31, ret = 0;

  for (int i = 0; i < 32; ++i) {
    if (n & 1)
      ret |= mask;
```

```
    mask >>= 1;
    n >>= 1;
  }

  return ret;
}
```

Solution 3: write bit at rightmost bit and left shift result In this approach, after we detect the bit value in input number, we set the bit at rightmost of our container as the result, then we perform a left shift to "push" the result to left. Repeat this until we extract all bits from the input number. Notice that we don't do left shift when the last bit is set (since it is the last one). In practice you can left shift `ret` first and then set value at the rightmost bit.

Code:

```
uint32_t reverseBits(unit32_t n) {
  unsigned int ret = 0;

  for (int i = 0; i < 32; ++i, n >>= 1) {
    ret <<= 1;
    ret |= n & 1;
  }

  return ret;
}
```

1.3.7 & tricks

The & operator can be used to select bits on certain position. Some numbers with special binary pattern are listed below (with their hexadecimal value, which is easier to remember):

Hex	Bin
0xaaaaaaaa	1010 1010 1010 1010 1010 1010 1010 1010
0x55555555	0101 0101 0101 0101 0101 0101 0101 0101
0xcccccccc	1100 1100 1100 1100 1100 1100 1100 1100
0x33333333	0011 0011 0011 0011 0011 0011 0011 0011
0xf0f0f0f0	1111 0000 1111 0000 1111 0000 1111 0000
0x0f0f0f0f	0000 1111 0000 1111 0000 1111 0000 1111
0xff00ff00	1111 1111 0000 0000 1111 1111 0000 0000
0x00ff00ff	0000 0000 1111 1111 0000 0000 1111 1111
0xffff0000	1111 1111 1111 1111 0000 0000 0000 0000
0x0000ffff	0000 0000 0000 0000 1111 1111 1111 1111

If a number & with the above special number, only bits coincide with the pattern's bits will be kept.

1.3.8 Bitwise AND of Numbers Range

Given a range `[m, n]` where `0 <= m <= n <= 2147483647`, return the bitwise AND of all numbers in this range, inclusive. For example, given the range `[5, 7]`, you should return `4`.

The naive solution is:

```
int rangeBitwiseAnd(int m, int n) {
  int ret = ~0;

  for (int i = m; i <= n; i++)
    ret &= i;

  return ret;
}
```

however the problem requires a faster algorithm.

Let's see an example, `m = 25`, `n = 29`:

```
25: 11001
26: 11010
27: 11011
28: 11100
29: 11101
```

When performing bitwise AND among all the numbers in the range, if any bit is 0, then the final result would be zero.

When number changes, the bit-value changes, it may be 0 or 1. We first find out what doesn't change from `m` to `n`:

```
25: 11...
26: 11...
27: 11...
28: 11...
29: 11...
```

The left two bits remain unchanged for all numbers in the range, so for these two bits:

```
25 & 26 & 27 & 28 & 29 == 25 & 29
```

For all the rest three bits, no bit can remain a constant 1 or 0 in the range (otherwise, it would be categorized as unchanged bit also). So the bitwise AND is zero (at least one zero would be in any bit position). Thus, the result of bitwise AND to all numbers in the range `[25, 29]` is `11000`.

So the solution is straightforward, first we keep right shifting until `m == n`, we count how many bits we shifted during the process. Then, we left shift back the same number of bits.

Code:

```
class Solution {
public:
  int rangeBitwiseAnd(int m, int n) {
    int a = 0;

    while (m != n) {
      m >>= 1;
      n >>= 1;
      a++;
    }
    return n << a;
  }
};
```

1.3.9 Repeated DNA sequences

This example will show how to create integer hash key for certain objects (of those related to integer type, like `char` or `string`) using bit manipulation.

The problem statement can be found here. One naive solution is to traverse the string, for each 10-char sequence, we store it in a hash table (`unordered_map<string, int>`). The value of the hash table is the appearing times of the char sequence. Then, we traverse the hash table, push each string (the key) to the result vector if the appearing time is more than 1.

Here we use another approach. We write the hash rule to map a 10-char long string to an integer. The 10-char string can only pick four possible characters: `A, T, G, C`. So for a character in the string, there are four states. To describe four states, we need 2 bits (00, 01, 10, 11). We define a map rule that translates `A, T, G, C` to four numbers `00, 01, 10, 11`:

```
int chtonum(char ch) {
  switch (ch) {
  case 'A':
    return 0;
  case 'T':
    return 1;
  case 'G':
    return 2;
  case 'C':
    return 3;
  default:
    return -1;
  }
}
```

As an example, following is the process to encode `s = "ATCG"`:

```
int key = 0;
for (int i = 0; i < 4; i++) {
  key = (key << 2) | chtonum(s[i]);
}
```

```
"A": 00
"AT": 00 01
"ATC": 00 01 11
"ATCG": 00 01 11 10
```

When the length of key reaches 20-bit (which means it encoded 10 chars), the next char can make it longer than 20-bit, to keep the length of key to 20-bit, we can use a mask:

```
int mask = 0xfffff;  // mask: 1111 1111 1111 1111 1111
```

so, after we update the `key`, we perform a bitwise AND with the mask, to keep the length of key as 20-bit:

```
key = ((key << 2) | chtonum(s[i])) & mask;
```

Now, we encoded the 10-char string (use a 20-bit integer to represent a unique combination of 10-char sequence). There are a total of 4^10 combinations of these sequences, so the number of potential keys is also 4^10. We can use an array to record each key's frequency, so we need to create an array big enough to hold all of it. The index is the possible key value.

```
int key_count[1 << 20] {0};
```

Each time we obtained a newly generated key, we check the value of key_count[key]. If it is 1, it means it already appeared before, we find out the 10-char string and push it to result vector. If it is other value (0 or > 1), it means it is either not appeared before, or appeared more than once (we have already pushed it in result vector, so we don't push it again). To find out the 10-char, just notice that the current char is the last char of this 10-char string, so the string is `s.substr(i - 9, 10)`.

Code:

```
class Solution {
public:
  // mapping from ch to integer number
  int chtonum(char ch) {
    switch (ch) {
    case 'A':
      return 0;
    case 'T':
      return 1;
    case 'G':
      return 2;
    case 'C':
      return 3;
```

```
    default:
      return -1;
    }
  }

  vector<string> findRepeatedDnaSequences(string s) {
    int length = s.size();

    if (length < 11)
      return {};

    int key = 0;  // used to store hash key for the 10-char string

    int mask = 0xfffff;
    vector<string> v;

    int key_count[1 << 20] {0};

    // calculate the key for the first 9 characters
    for (int i = 0; i < 9; i++)
      key = (key << 2) | chtonum(s[i]);
    // find the possible sequences in rest of the string
    for (int i = 9; i < length; i++) {
      key = ((key << 2) | chtonum(s[i])) & mask;
      key_count[key]++;
      if (key_count[key] == 2) {
        v.push_back(s.substr(i - 9, 10));
      }
    }

    // return the result vector
    return v;
  }
};
```

1.3.10 Majority Element

The problem statement can be found here. Other solutions possible (check my leetcode notes). Here we use bit manipulation to solve it.

Take array `[5,5,4,5,1,5,6,5]` as an example. the binary format of those numbers are listed below:

```
5: 0101
5: 0101
4: 0100
```

```
5: 0101
1: 0001
5: 0101
6: 0110
5: 0101
```

Since the majority element if the one that appears more than `n / 2` times, the bit value of the majority element should also be "majority bit value". So we just need to find out the majority bit value at each bit, and reconstruct the majority element using this information. In the above example, the majority bit value for each bit (from left to right) i s: `0, 1, 0, 1`, thus the majority element is `0101`, which is 5.

Code:

```cpp
class Solution {
public:
  int majorityElement(vector<int>& nums) {
    int length = 8 * sizeof(int);
    int size = nums.size();
    int count = 0;
    unsigned int mask = 1;
    int ret = 0;

    for (int i = 0; i < length; i++) {  // traverse over each bit
      count = 0;  // count number of 1-bit at current bit (for all number
      ↪  in nums)
      for (int j = 0; j < size; j++)
        if (mask & nums[j])
          count++;
      if (count > size / 2)  // if number of 1-bit is greater than size /
      ↪  2, it is the majority bit, update directly (better than my
      ↪  solution). You only consider 1-bit case, because ret is 0-bit by
      ↪  default
        ret |= mask;

      mask <<= 1;
    }

    return ret;
  }
};
```

1.3.11 Single number II

The problem statement can be found here.

Bit counting For each bit position, count the number of 1-bit on the position. If the single number has a 0-bit here, the total bit_count should mod 3 (because other numbers appear exactly three times). If the single number has a 1-bit here, `bit_count % 3 != 0`. We can use this to determine bit value of the single number at each bit and reconstruct it during the process.

Code:

```
class Solution {
public:
  int singleNumber(vector<int>& nums) {
    int length = 8 * sizeof(int);
    int ret = 0;
    unsigned int mask = 1;
    int bit_count;

    // for each bit, calculate bit_count for all numbers
    for (int i = 0; i < length; i++) {
      bit_count = 0;

      for (int num : nums)
        if (num & mask)
          bit_count++;

      if (bit_count % 3)
        ret |= mask;

      mask <<= 1;
    }

    return ret;
  }
};
```

Logical circuit design Not quite understand. Put it to future work.

Check other solutions on the website.

1.3.12 Single number III

The problem statement can be found here.

You are given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

Example:

```
input: [1,2,1,3,2,5]
Output: [3,5]
```

To cancel number that appeared twice, we can think of using exclusive or, because `a ^ a == 0`. Assume the two single number is `a` and `b`. We traverse the array first to use `0` to XOR each number, we'll get `a ^ b`.

How to find out `a` and `b` using `a ^ b`? By taking a look at the bits in `a ^ b`, we found that any 1-bit in this number suggests `a` and `b` has different bit-value at this bit-position. By using this information, we can divide the array into two groups: all numbers has a 1-bit at certain bit position (of those 1-bit position in `a ^ b`) goes to group A, all number has a 0-bit at the same bit-position goes to group B. So the two single numbers are separated in group A and group B. If we do another xor for each number in two groups, we can get the individual value of `a` and `b`.

There is not need to use extra space to store two groups, we xor each number when it is grouped.

This solution was provided by this reference. Code:

```
vector<int> singleNumber(vector<int>& nums) {
  int mask;
  int axorb = 0;
  int a = 0;
  int b = 0;

  // get a ^ b
  for (int num : nums)
    axorb ^= num;

  // find out different bit in a and b
  mask = axorb ^ (axorb & (axorb - 1));

  // go over array again and axorb two groups
  for (int num : nums) {
    if (num & mask)
      a ^= num;
    else
      b ^= num;
  }

  return {a, b};
}
```

1.3.13 Maximum product of word lengths

The problem statement can be found here.

The tricky part is how to determine two words intersect or not.

If using brutal force, for two words `A` and `B`, we start with the first letter in `A`, traverse `B` to find out if `B` contains the letter or not. Then, we do the same thing for the second letter in `A`. Assume the average length of each word is `k`. We need `k^2` comparisons just to determine if `A` and `B` intersects or not. And we must do this for any pair of two words (which is Cn2). We need to find a way that can determine whether two words intersects or not faster.

Bitwise AND operator is a fast way to perform intersection on two numbers (two sets). For a word and its letters, you may want to design a method to transform a word into an integer, the bit-value in this integer should carry information of which letter appeared in the original word. When we intersect two such integers, the result can show us wether the two words that behind this two integers have common letters or not.

Notice that there is no limit to the word length. Also, if a character repeats itself in a word, our encoded integer should encode just one character (because this is enough).

At first, I was thinking to encode each letter to a series of bit. There are 26 letters, so there are 26 states, I need a minimum of 5 bits to describe them all (because `2^5` $>$ 26). But an integer only has 32 bits, which means if using this method, I can only encode a word with length less than 6 letters (record which word appeared in this word). How to encode a word to an integer?

In fact, you don't have to use 5-bit to record a single character's appearance. The integer type has a natural structure which we can use to map to 26 letters. That is the bit position. An integer has 32 bits, we can use the first 26 bits to indicate 26 letters. For example:

```
zyxwvutsrqponmlkjihgfedcba
00000000000000000000000000
```

The bit value at each bit can be either 1 or 0, we use this to indicate whether the corresponding letter appeared or not. For example, to encode `"abce"`, we have:

```
zyxwvutsrqponmlkjihgfedcba
00000000000000000000010111
```

So, the steps are:

1. transform the word list to an integer list, which contains info of letter appearing frequency
2. traverse this integer array, for each word, check if there is any word after intersects with it. If so, calculate product of word length and update the running maximum product. No need to check backward because they are already checked in previous run of the loop.

Some trival details that can enhance your code:

1. use an array to store the length of words. So no need to call .length() frequently
2. use max() function to determine which one is larger.

1.4 place holder