LC 461. Hamming Distance

```
class Solution {
public:
   int hammingDistance(int x, int y) {
      int count = 0;

      for (int n = x ^ y; n != 0; n &= n - 1)
         ++count;

      return count;
   }
};
```

LC 190. Reverse Bits

```
class Solution {
public:
   uint32_t reverseBits(uint32_t n) {
      uint32_t rev = 0;
      int i = 0;

      while (n != 0)
      {
         rev <<= 1;
         rev |= n & 1;
         n >>= 1;
         ++i;
      }

      return i == 0? rev : rev << (32 - i);
   }
};
```

LC 187. Repeated DNA Sequences

```
class Solution {
public:
    /**
     * @param s: a string
     * @return: return List[str]
     */
    vector<string> findRepeatedDnaSequences(string &s) {
        constexpr int len = 10;

        // Corner case
        if (s.size() <= len) return {};

        // Initialization
        uint32_t code = 0;

        for (auto i = s.cbegin(); i != s.cbegin() + len; ++i)
            encodeDna(code, *i);

        unordered_map<uint32_t, int> count{ {code, 1} };
        vector<string> ans;

        // Check each sequence
        for (auto i = s.cbegin() + len; i != s.cend(); ) {
            encodeDna(code, *i++);
            if (count[code]++ == 1) ans.emplace_back(i - len, i);
        }

        return ans;
    }

    void encodeDna(uint32_t& n, const char& c) {
        n <<= 2;

        if      (c == 'C') n += 1;
        else if (c == 'G') n += 2;
        else if (c == 'T') n += 3;

        n &= 0xFFFFF;
    }
};
```

LC 201. Bitwise AND of Numbers Range

```
class Solution {
public:
    /**
     * @param m: an Integer
     * @param n: an Integer
     * @return: the bitwise AND of all numbers in [m,n]
     */
    int rangeBitwiseAnd(int m, int n) {
        int count = 0;

        while (m != n) {
            m >>= 1;
            n >>= 1;
            ++count;
        }

        return m << count;
    }
};
```