< Back | All Types of Patterns for Bits Manipulations and How to use it

imanishkumar7545 ★ 3104 Last Edit: 2 days ago 2.5K VIEWS

141

Bitwise Operators: Manipulating Binary Numbers

Bitwise operators are powerful tools for manipulating binary numbers. They operate at the bit level, allowing you to perform various operations efficiently. Here are the most common bitwise operators and their functionalities:

NOT Operator (~)
The NOT operator, represented by the tilde (`~`) symbol, flips the bits of a binary number. It changes each 0 to 1 and each 1 to 0. This operator is useful for inverting the bits of a number.

AND Operator (&)
The AND operator, denoted by the ampersand (`&`) symbol, compares two binary numbers bit by bit. It sets each bit to 1 only if both corresponding bits are also 1. Otherwise, it sets the bit to 0. The AND operator is commonly used for masking and checking the presence of specific bits.

OR Operator (|)
The OR operator, represented by the vertical bar (`|`) symbol, compares two binary numbers bit by bit. It sets each bit to 1 if either of the corresponding bits is 1. If both bits are 0, it sets the resulting bit to 0. The OR operator is useful for combining or merging bits.

XOR Operator (^)
The XOR operator, denoted by the caret (`^`) symbol, compares two binary numbers bit by bit. It sets each bit to 1 if exactly one of the corresponding bits is 1. If both bits are the same (either both 0 or both 1), it sets the resulting bit to 0. The XOR operator is often used for toggling or swapping bits.

Left Shift Operator (<<)
The left shift operator (`<<`) shifts the bits of a binary number to the left by a specified number of positions. This operation is equivalent to multiplying the number by 2 raised to the power of the shift amount. It is frequently used for efficient multiplication or creating space for additional bits.

Right Shift Operator (>>)
The right shift operator (`>>`) shifts the bits of a binary number to the right by a specified number of positions. This operation is equivalent to dividing the number by 2 raised to the power of the shift amount. It is commonly used for efficient division or extracting specific bits.

With these bitwise operators, you can perform complex operations on binary numbers, manipulate individual bits, and optimize certain calculations efficiently.

Please Pay some attention Here to understand bits in better manner
Here i have shared difference between binary and decimal number to uderstand what Bits is actually

Feature	Decimal	Binary
Base	`10`	`2`
Possible digits	`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`	`0, 1`
Place values	Powers of `10`	Powers of `2`
Example	`123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0 = 100 + 20 + 3` = 123	`5 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 101`

The main difference between binary and decimal number systems is the **base**. Binary is a base-2 number system, while decimal is a base-10 number system.*This means that the possible digits in binary **are limited to 0 and 1**, while the possible digits in decimal can be any number from **0 to 9**.*

The place values in binary are also different from the place values in decimal. In binary, the place values are powers of 2, while the place values in decimal are powers of 10. This means that the value of a digit in binary depends on its position in the number, just like the value of a digit in decimal.

When you ask me to print the binary of 14, I will print 1110. This means that the value of 14 in binary is 1110. The digit 1 in the 3rd place represents the value of 2^3, which is 8. The digit 1 in the 2nd place represents the value of 2^2, which is 4. The digit 1 in the 1st place represents the value of 2^1, which is 2. And the digit 0 in the 0th place represents the value of 2^0, which is 1 so 14=*12^3+12^2+12^1+02^0*

In decimal, we do the same thing, but we don't usually pay attention to it because it is so natural. For example, the number 1234 is equal to **1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0**. The digit 1 in the 3rd place represents the value of 10^3, which is 1000. The digit 2 in the 2nd place represents the value of 10^2, which is 200. The digit 3 in the 1st place represents the value of 10^1, which is 30. And the digit 4 in the 0th place represents the value of 10^0, which is 4.

So, when you ask me to print the binary of 14, **I am telling you the value of each power of 2 in the number**. The same principle applies to decimal numbers. When you ask me to print 1234 in decimal,**I am telling you the value of each power of 10 in the number.**

Lets understand how to seprate each digit from each place of decimal
For example, to separate the digits from the number 1234

Decimal number	Digit	Operation
1234	4	1234 % 10 = 4
123	3	floor(123 / 10) = 123 %10 = 3
12	2	floor(12 / 10) = 12 %10 = 2
1	1	floor(1 / 10) = 1 % 10 = 1

In the table, each digit is nothing but the decimal value corresponding to some power of 10. For example, the digit 4 in the number 1234 represents the value of 10^0, which is 1. The digit 3 in the number 1234 represents the value of 10^1, which is 10. And so on.

The reason why the modulus operator works is because the remainder of a division is always the least significant digit of the dividend. The least significant digit is the digit with the smallest place value. In the decimal number system, the place values decrease by 1 as you move to the left. So, the units digit has the smallest place value, followed by the tens digit, then the hundreds digit, and so on.

Therefore, **The modulus operator can be used to extract the least significant digit of a number. This is useful for separating the digits of a number from each place.**

The same principle can be applied to binary numbers. The least significant bit of a binary number is the bit with the smallest place value, which is 2^0. The next bit is 2^1, and so on.

To extract the least significant bit of a binary number, **we can use the modulus operator**. For example, the least significant bit of the number 1010 is 0, because 1010 % 2 = 0.

To extract the next bit, we can divide the number by 2 and then take the floor of the result. So, the next bit of 1010 is 1, because floor(1010 / 2) = 505 // 2 = 252.

We can continue this process to extract all of the bits in a binary number.

Bit	Operation	Value
0	number % 2	The least significant bit of the number
1	floor(number / 2)	The next bit of the number
2	floor(number / 4)	The next bit of the number
3	floor(number / 8)	The next bit of the number
...	...	...

eg-> 15

Bit	Operation	Value
0	15 % 2	1
1	floor(15 / 2)=7%2	1
2	floor(7 / 2) =3 %2	1
3	floor(3 / 2)=1%2	1
4	floor(1 / 2) =0 stop here	As no set set bit is left

May you get the idea of Decimal and Binary number how they related to each other

Attention

```
Result after shifting left(or right) too much is undefined
Right shifting operations on negative values are undefined
Right operand in shifting should be non-negative, otherwise the result is undefined
The & and | operators have lower precedence than comparison operators
```

Basic you Must have to know

1) Printing the binary representation of any Number.

```
void pr_binary(int num){
    for(int i=10;i>=0;i--) cout<<((num>>i)&1);
    cout<<endl;
}
```

2) checking if the ith bit is set or not.

```
if((num&(1<<i))!=0) cout<<"set"<<endl;
  // check if set or not;
  else cout<<"Not set"<<endl;
```

3) Counting the number of set bits

```
  int count=0,num=15;
  for (int i=31; i>=0;--i)
  if((num&(1<<i))!=0) count++;
```

4) How to check if a given number is a power of 2 ?

```
Properties for numbers which are powers of 2, is that they have one and only
one bit set in their binary representation.
If the number is neither zero nor a power of two, it will have 1 in more than
one place. So if x is a power of 2 then x & (x-1) will be 0.
eg.-> 16-> 10000 and 15 -> 1111
16&15==0

 bool isPowerOfTwo(int x)
  {
    return (x && !(x & (x - 1)));
  }
```

5) Dividing or multiplying any number by two

```
 //Although the arithmetic operations are fast ,but by bits manipulation
// we can make them  more faster.
  int n=5;
  n=n>>1;
 // divide by two
  n=n<<1;
 // multiply by two
```

6) Some cool operations and playing with Characters

```
Difference between upper case letter and lower case letter binary is that
 In upper case letter 5th bit!=1;
 In lower case letter 5th bit =1;
 cout<<char('A'|(1<<5))<<endl;     //Convert Char to lower case;
 cout<<char('a'&(~(1<<5)))<<endl;    //Convert Char to upper case;

 Actually char of 1<<5 is _(space);
 Take any upper case letter and its |(or) with space will get the corresponding lower case letter;

 cout<<char('C'|' ')<<endl;   // will make it small c

Take any lower case letter and its &(and) with _(underscore) will get the corresponding upper //case letter;`

 cout<<char('c'&'_')<<endl;   // will make it capital C
```

```
Find a letter's position in alphabet
We can easily find a letter's position [1-26] in the
alphabet by taking its bitwise AND with ASCII 31 (00011111 in binary).
The case of the letter is irrelevant here.

 eg.
 ('A' & 31) returns position 1
 ('c' & 31) returns position 3
```

7) Swap with XOR.

```
  int a=4;
  int b=5;
  a=a^b;
  b=b^a;
  a=a^b;
  // cout<<a<<" "<<b;
```

8) For clearing the set bits upto ith bit

```
 int i=4;
//clearing upto 5 the place;
int a=59;
int b=(a&(~((1<<(i+1))-1)));
//clearing the lsb upto ith bit;


i=3;
int c=(a&((1<<(i+1))-1));
//clearing the msb upto ith bit;
```

Some Medium level uses

```
Set union A | B
Set intersection A & B
Set subtraction A & ~B
Set negation ALL_BITS ^ A or ~A
Set bit A |= 1 << bit
Clear bit A &= ~(1 << bit)
Test bit (A & 1 << bit) != 0
Extract last bit A&-A or A&~(A-1) or x^(x&(x-1))
Remove last bit A&(A-1)
Get all 1-bits ~0==-1
```

Two's Complement

In two's complement, **the MSB is a sign bit**.
MSB Most significant bit or You can Say left most bit
A 0 in the MSB indicates a positive number while a 1 indicates a negative number.
so the most significant bit (MSB) is used to indicate the sign of the number.

The remaining 31 bits are used to represent the magnitude of the number.(I am taking 32bit number)
For example, the number *1 is represented as 00000000000000000000000000000001*.
For example, the number *-1 is represented as 11111111111111111111111111111111*.

In other words, the negative number -1 is represented as the bitwise complement of the positive number 1, plus 1.
The bitwise complement of a number is the number obtained by flipping all of the bits in the number. So, the bitwise complement of 1 is 11111111111111111111111111111110. When we add 1 to this number, we get 11111111111111111111111111111111, which is the two's complement representation of -1.

SO a negative number is equal to the bitwise complement of its positive counterpart, plus 1.
negative_number = ~positive_number + 1
In two's complement representation, subtraction can be done by simply adding the two numbers together, and then taking the bitwise complement of the result

```
From Here we can say
Num= num1-num2;
Num= (num1)+((~num2)+1)
```

Now suppose we want to subtract 12 from 69

```
Binary representation of 69: 00000000000000000000000001000101
Binary representation of -12: 11111111111111111111111111110100
(obtained by inverting and adding one)

Carry Row:
 00000000000000000000000001000101 (69)
+ 11111111111111111111111111110100 (-12)
 --------------------------------
 00000000000000000000000000111001 (57)
```

Important Patterns For BitsManupliation

Pattern->1.Gray code

```
Binary Number to gray Code
 vector<int> grayCode(int n)
   {
      vector<int>v;
      for(int i=0 ;i<(1<<n);i++)
      v.push_back(i^(i>>1));
      return v;
   }
```

```
Gray Code to Binary Number
int rev_grayCode (int g) {
 int n = 0;
 for (; g; g >>= 1)
   n ^= g;
 return n;
}
```

Questions for gray code

1. gray-code
2. circular-permutation-in-binary-representation

Pattern->2.Checking is power of something eg ->4

```
Q ) Check if a number is a power of 4 or not  ?
The given number n is a power of 4 if it is a power of 2 and
its remainder is 1 when it is divided by 3.
bool checkPowerOf4(unsigned n)
{
    // return true if `n` is a power of 2, and
    // the remainder is 1 when divided by 3
    return !(n & (n - 1))&& (n % 3 == 1);
}
```

Questions

1. Power-of-two
2. Power-of-three
3. power-of-four

Pattern 3 -> Question on the Basic Prop of xor

```
1) xor of a same number with itself is zero, i.e A ^ A = 0
2) xor is commutative that means a ^ b = b ^ a.
3) xor of any number with zero is the number itself i.e A ^ 0 = A.
```

Questions:

1. single-number
2. single-number-ii
3. single-number-iii
4. find-the-original-array-of-prefix-xor
5. count-triplets-that-can-form-two-arrays-of-equal-xor
6. xor-queries-of-a-subarray

Pattern 4 ->Based On fact Seting Left most bit give You Large Number
Eg - for four bit
0111 < 1000 Always
So based On this fact many Question is asked
Questions

1. maximum-xor-of-two-numbers-in-an-array

```
I have given code here Because i have used same concept Here.
When i am reading solution section i just not found any solution like this
most of them  just use tries  SO think it should be shared
class Solution {
public:
    void solve(int target,vector<int>&nums,int &ans)
    {
        unordered_set<int>s;
        for(auto dig:nums)
        {
             auto digit=((target&dig));
            if(s.count(digit^target)!=0)
            {
               ans=target;
               return ;
             }
              s.insert(digit);
        }
    }
    int findMaximumXOR(vector<int>& nums)
    {
      int ans=0;
      for(int i=31;i>=0 ;i--)
      solve((ans| (1<<i)),nums,ans);
      return ans;

    }
};

/*
Logic: for a 32 bit number we have 32 bits in memory, so in order to
find the largest possible value of an XOR operation, the value of XOR
should have most of the bits set (i.e. 1) starting from the left to right
*/
```

2. score-after-flipping-matrix

Pattern 5 ->DP + Bitmasks
This is actually a very well-known technique and most people should already know this. This trick is usually used when one of the variables have very small constraints that can allow exponential solutions.
Usually, when doing DP + Bitmasks problems, we store the subsets as an integer from 0 to 2n - 1
I have added a standard format You follow your own

```
 vector<vector<int>>dp;
int Generate_subset(int ind,int mask, int n)
{
    if(ind>=n){
        return 1;
    }
    if(dp[ind][mask]!=-1)
    return dp[ind][mask];

    int ans=0;
    for(int i=0;i<n;i++)
    {
        if((1<<i&mask))
        continue;
        if(Condition is based on questions)
     // Generally you have to change just condition here and you will get answer
        ans=Some Operations (ans,Generate_subset(ind+1,(1<<i|mask),n));

    }
    return dp[ind][mask]=ans;
}
```

Eg. On above format
Soln-> minimum-xor-sum-of-two-arrays
You may confused about how to cover all permutations in 2^n time if there are N factorial combinations. Let's take an example to understand this.

---

nums1 = [1, 2, 3, 4, 5, 6, 7] and nums2 = [8, 9, 10, 11, 12, 13, 14]

What drives the bits?
The number of permutations of nums2 is 7! = 5040. However, we can use bitmasks to reduce the number of permutations that we need to consider to 2^7 = 128.
A bitmask is a binary number that represents the elements of an array that have been used so far.
The number of permutations of the second array is 7! = 5040. However, the number of states that we need to consider is only 2^7 = 128. This is because each state represents a different prefix of the second array.
For example, the bitmask 0000111 represents the elements 8, 9, ans 10.The bitmask 0001001 represents the elements 8, and 12. And so on.

Why is this important?
This is important because it allows us to reduce the number of permutations that we need to consider. If we did not use bitmasks, we would need to consider all 7! = 5040 permutations of the second array. However, by using bitmasks, we only need to consider 2^7 = 128 permutations. This is a significant speedup, and it allows us to solve the problem in a reasonable amount of time.
Dry run if Not get my point

```cpp
class Solution {
public:
    vector<vector<int>>dp;
    int dfs(int idx,int mask,vector<int>&nums1,vector<int>&nums2)
    {
        if(idx>=nums1.size())
        return 0;
        if(dp[idx][mask]!=-1)
        return dp[idx][mask];
        int ans=INT_MAX;
        for(int i=0 ;i<nums2.size();i++)
        {
          int bits= (1<<i);
          if(bits&mask)
          continue;
          ans= min (ans ,(nums1[idx]^nums2[i])+dfs(idx+1,mask|bits,nums1,nums2));
        }
        return dp[idx][mask]=ans;
    }
    int minimumXORSum(vector<int>& nums1, vector<int>& nums2)
    {
        dp.resize(nums1.size()+1 ,vector<int>(1<<nums2.size()+5,-1));
        return dfs(0,0,nums1,nums2);
    }
};
```

Questions:

1. 526. Beautiful Arrangement
2. 698. Partition to K Equal Sum Subsets
3. 2572. Count the Number of Square-Free Subsets
4. 2035. Partition Array Into Two Arrays to Minimize Sum Difference
5. 1659. Maximize Grid Happiness
6. 1723. Find Minimum Time to Finish All Jobs (Exact same as Fair-Distribution of cookies)
7. 1255. Maximum Score Words Formed by Letters
8. special-permutations
9. subsets
10. subsets-ii
11. Fair-distribution-of-cookies
12. Minimum-number-of-work-sessions-to-finish-the-tasks

4. [2035. Partition Array Into Two Arrays to Minimize Sum Difference]
The above question is actually special one here you will learn about
Meet in the middle So must do This

Soln of Problem 2035

Out of an array of length N, we first partition it into left and right arrays,
each of length N/2. This partitioning technique is similar to divide and conquer.

Step 0: Pre-compute all the subarray lengths and their possible sums on either side of N/2.

Example: Let's consider the array nums = [3, 9, 7, 3]

Left (0 to N/2-1):

Subarray Length	Possible Sums
0	[0]
1	[3, 9]
2	[12]

Right (N/2 to N-1):

Subarray Length	Possible Sums
0	[0]
1	[7, 3]
2	[10]

Now, let's consider what we are going to do. We can take each subarray of length i
from the left array and compare it with each subarray of length N-i from the right array.

To optimize this process, we can utilize the `lower_bound` operation and
`totalSum/2 - iLengthSubArrSum`. Without these optimizations, the
time complexity would be O(N^2) due to the nested loops and the search operation.
However, these optimizations help improve the efficiency.

```
code :
vector<vector<int>> findAllSubsetsSum(vector<int>& nums, int l, int r) {
    int totLengthOfSubarray = r - l + 1;
    vector<vector<int>> res(totLengthOfSubarray + 1);
    for (int i = 0; i < (1 << totLengthOfSubarray); i++) {
        int sum = 0, countOfChosenNos = 0;
        for (int j = 0; j < totLengthOfSubarray; j++) {
            if (i & (1 << j)) {
                sum += nums[l + j];
                countOfChosenNos++;
            }
        }
        res[countOfChosenNos].push_back(sum);
    }
    return res;
}

int minimumDifference(vector<int>& nums) {
    int totalSum = accumulate(begin(nums), end(nums), 0);
    int n = nums.size();

    auto left = findAllSubsetsSum(nums, 0, n / 2 - 1);
    auto right = findAllSubsetsSum(nums, n / 2, n - 1);
    int target = totalSum / 2, ans = INT_MAX;

    //we can take (0 to n/2) length numbers from left
    for (int i = 0; i <= n / 2; i++) {
        //now we take rest - (n/2-i) length from right, we sort it to binary search
        auto r = right[n / 2 - i];
        sort(begin(r), end(r));

        for (int curleftSum : left[i]) {
            int needSumFromRight = target - curleftSum;
            auto it = lower_bound(begin(r), end(r), needSumFromRight);
            if (it != end(r))
                ans = min(ans, abs(totalSum - 2 * (curleftSum + *it)));
        }
    }
    return ans;
}

This is code is actually of  @geekykant
```

Pattern 6 -> Restricting Over some Operation e.g. -> + and - and / etc
You should Practice some Questions to solve these Type of patterns.
So I have added some link :
Questions:

1. Divide two integers without using multiplication, division, and mod operator.
2. Multiply two numbers without using multiplication(Search for it)
3. Implement power function without using multiplication and division operators

```
The idea is if x = ab, then log(x) = b.log(a).
Since, x can be expressed as x = elog(x), by substituting the value of log(x)
in the equation, we get x = eb.log(a).
int pow(int a, int b)
{
    float logx = 0;
    for (int i = 0; i < b; i++) {
        logx += log(a);
    }

    return exp(logx);
}
```

4. Product of Array Except Self

Pattern 7 -> We can Hash any string to bit mask .
We can use this trick to reduce the constant factor of 26 in many different kind of problem.

```
int count(string & s) {
   int mask = 0;
   for (char i : s) {
    mask |= (1 << (i - 'a'));
   }
   return __builtin_popcount(mask);
}
```

e.g->

1. maximum-product-of-word-lengths
   If you will try to solve this question using boolean array the time complexity will be O(nn26) that will lead to tle but you can improve time complexty to O(n*n) just by using bitmask

```
public:
    int maxProduct(vector& words) {
        vector a(words.size() , 0);
        for(int i = 0;i < words.size();i++){
            for(int j = 0;j < words[i].size();j++){
                a[i] |= 1 << (words[i][j] - 'a');
            }
        }
        int res = 0;
        for(int i = 0;i < words.size();i++){
            for(int j = i + 1;j < words.size();j++){
                if((a[i] & a[j]) == 0){
                    if(words[i].size() * words[j].size() > res){
                        res = words[i].size() * words[j].size();
                    }
                }
            }
        }
        return res;
    }
};
```

Pattern 8-> Decode XORed Array.
In This type of Problems generally you have given a array and based on props of xor
You have to decode original array
Generally taking xor off all element give you answer or same time us prop of a^a==0
or A^B=c then A=B^c
Questions

1. Decode-xored-permutation
2. Decode-xored-array
3. Find-xor-beauty-of-array
4. find-the-original-array-of-prefix-xor

Pattern 9-> Generating submasks of a bitmask
Mask is nothing but a specific pattern of bits you use for some desired operation. For eg. to turn on 3th bit(0 based indexing) of a given number, you can use the mask 1000 (8 in decimal) and OR with the given number.

A submask is a mask whose set bits are a subset of the set bits of the original mask. In other words, a submask is a mask that can be obtained by turning off some of the bits of the original mask.

For example, the mask 1011 has three set bits: the first, second, and third bits. The masks 1010, 1000, and 0010 are all submasks of 1011, because they all have the same three set bits. The mask 1100 is not a submask of 1011, because it has a set bit in the fourth position, which 1011 does not have.

We know that subtracting 1 from a given number unsets the rightmost set bit and sets all the unset bits after it.
Inother word num- 1, it flips all the bits present on the right of the rightmost set bit including rightmost set bit of num
For example, if the mask is 10101000, then the mask after subtracting 1 will be 10100111. The rightmost set bit in the original mask was the 3rd bit from the right, so all the bits to the right of that bit are flipped including that bit.

We will use this to print all submask .If we subtract 1 from our mask and then AND with the original mask, we will end up with a submask with it's rightmost set bit unset
If you didn't understand, give it a dry run

```
void generate_submasks(int mask)
{|
  int submask=mask;
  while(submask)
  {
    cout << submask << endl;
    submask=mask&(submask-1);
  }
  cout << submask << endl; // Zero is also a valid submask
}
```

All props of num and (num-1) Note:-> In code i have refer num as mask

```
Note: These are some cool operations that you can do with (mask)&(mask-1).
All of them are discussed in detail in this article, but I have added them here to remind
you that they are important.

1.mask&(mask-1) can be used to remove the rightmost bits from a mask.
2.mask &(mask-1) or mask==0 can be used to check if a number is a power of two.
3.mask&~(mask-1) or (mask^(mask&(mask-1)) or (mask&-mask) can be used to extract the last bit of a number.
eg->    1010 &~(1010 -1)
         1010 & ~(1001)
         1010 &(0110)
         0010 ( return  The last  set bit )
4:->Brian Kernighan's Algorithm to count set bits in an integer
int countSetBits(int n)
{
    // `count` stores the total bits set in `n`
    int count = 0;
    while (n)
    {
        n = n & (n - 1);    // clear the least significant bit set
        count++;
    }
    return count;
}
5:->To generate all of the submasks
void generate_submasks(int mask)
{|
  int submask=mask;
  while(submask)
  {
    cout << submask << endl;
    submask=mask&(submask-1);
  }
  cout << submask << endl; // Zero is also a valid submask
}
```

Questions:

1. Number-of-valid-words-for-each-puzzle
2. Fair Distribution of Cookies
   (Soln.)DP | Submask Enumeration
3. Minimum-incompatibility
4. minimum-cost-to-connect-two-groups-of-points
   Soln) using-bitmask-and-submask-trick
5. Problems/parallel-courses-ii

Note: For subset dp using mask You can follow his article(*I think This is well explained*) :->
Dynamic-programming-on-subsets
Problems discussed in his article are listed below

1. Partition-to-k-equal-sum-subsets
2. Matchsticks-to-square
3. Beautiful-arrangement
4. Shortest-path-visiting-all-nodes
   Soln of Shortest-path-visiting-all-nodes Actually in leetcode sol i have not found in neet code thats why i have added here my code

```
class Solution {
public:
   int shortestPathLength(vector<vector<int>>& graph)
   {
       int n=graph.size();
       int all = (1<<n)-1;
       queue<vector<int>>q;
       set<pair<int,int>>vis;
       for(int i=0;i<n;i++)
       {
           int mask=(1<<i);
           q.push({i,0,mask});
           vis.insert({i,mask});
       }

      while(q.size())
      {
          auto node= q.front();
          q.pop();
          int parent=node[0],dist=node[1],mask=node[2];
          for(auto child:graph[parent])
          {
              int newMask= (mask | (1<< child));
              if(newMask==all)
              return dist+1;
              else if(vis.count({child,newMask})) continue;
              else
              {
                  q.push({child,dist+1,newMask});
                  vis.insert({child,newMask});
              }
          }
      }
      return 0;
   }
};
```

5. Find-the-shortest-superstring
6. Number-of-ways-to-wear-different-hats-to-each-other
7. Parallel-courses-ii
8. Distribute-repeating-integers
9. Maximize-grid-happiness
10. Minimum-incompatibility
11. k-similar-strings
12. Maximize-score-after-n-operations
13. Minimum-xor-sum-of-two-arrays
14. Maximum-students-taking-exam
15. Smallest-sufficient-team

Pattern 10-> Kth bits questions

1. Find-kth-bit-in-nth-binary-string
2. K-th-symbol-in-grammar

sol of Kth symbol-in grammar

The problem statement states that N and K are 1-indexed (although N is not required to solve this problem). To avoid ambiguity, we can easily convert K to a 0-indexed notation by subtracting 1 from K, denoting it as k. Hence, k = K - 1.

Now, you'll notice that finding the symbol for any k simply requires performing an XOR operation on its bits. For example, let's consider N = 4 and K = 5. In this case, we have k = K - 1 = 4. The binary representation of 4 is 100. XORing the bits results in 1, which is the correct symbol.

Another example would be N = 4 and K = 7. In this scenario, k = 6. The binary representation of 6 is 110. XORing the bits gives us 0.

```
class Solution{
public:
    int KthGrammar(int N, int K)
    {
        K -= 1;
        int val = 0;
        while(K)
        {
            val ^= (K&1);
            K >>= 1;
        }
        return val;
    }
};
```

Some Important Things You should Remember

1.We can find the minimum XOR of two elements in an array just by
sorting the array and taking the minimum of XOR-s of neighboring elements.
2.To find the maximum XOR of two elements we can use the trie data structure. [How?]
And One other Method i have discussed above
3.We can use bitmasks to solve problems related to the inclusion-exclusion principle

SOME MORE IMPORTANT HACKS:

```
x&1 gives the lowest bit(helps in finding whether number is even or odd
i.e if last bit is 0 then it is even otherwise odd)
x & (x-1) will clear the lowest set bit of x
x & ~(x-1) extracts the lowest set bit of x (all others are clear).
Pretty patterns when applied to a linear sequence.
x & (x + (1 0<< n)) = x, with the run of set bits (possibly length 0) starting at bit n cleared.
x & ~(x + (1 << n)) = the run of set bits (possibly length 0) in x, starting at bit n.
x | (x + 1) = x with the lowest cleared bit set.
x | ~(x + 1) = extracts the lowest cleared bit of x (all others are set).
x | (x - (1 << n)) = x, with the run of cleared bits (possibly length 0) starting at bit n set.
x | ~(x - (1 << n)) = the lowest run of cleared bits (possibly length 0) in x, starting at bit n are the only clear bits.
```

(Playing with k'th bit)

1.Turn off k'th bit in a number

```
int turnOffKthBit(int n, int k) {
    return n & ~(1 << (k - 1));
}
```

2.Turn On k'th bit in a number

```
int turnOnKthBit(int n, int k) {
    return n | (1 << (k - 1));
}
```

3.Check if k'th bit is set for a number

```
bool isKthBitSet(int n, int k) {
    return (n & (1 << (k - 1))) != 0;
}
```

4.Toggle the k'th bit

```
int toggleKthBit(int n, int k) {
    return n ^ (1 << (k - 1));
}
```

(Playing with the rightmost set bit of a number)
1.Find the position of the rightmost set bit

```
int positionOfRightmostSetBit(int n)
{
    // if the number is odd, return 1
    if (n & 1) {
        return 1;
    }

    return log2(n & -n) + 1;
}
Note: If (n & -n) == n, then the positive integer n is a power of 2.
```

3.Brian Kernighan's Algorithm to count set bits in an integer

```
The expression n & (n-1) can be used to turn off the rightmost set bit of a number n.
This works as the expression n-1 flips all the bits after the rightmost set bit of n,
including the rightmost set bit itself. Therefore, n & (n-1)
results in the last bit flipped of n.

int countSetBits(int n)
{
    // `count` stores the total bits set in `n`
    int count = 0;

    while (n)
    {
        n = n & (n - 1);    // clear the least significant bit set
        count++;
    }

    return count;
}
```

4.Find minimum or maximum of two integers without using branching

```
To find the minimum(x, y), we can use the expression y ^ ((x ^ y) & -(x < y)).
Now, if x is less than y, then -(x < y) will be -1
val= y ^ ((x ^ y) & -(x < y))
val = y ^ ((x ^ y) & -1)
val= y ^ (x ^ y)
val= x
If x is more than y, then -(x < y) will be 0
val = y ^ ((x ^ y) & -(x < y))
val = y ^ ((x ^ y) & 0)
val = y ^ 0
val = y

int findMin(int x, int y) {
    return y ^ ((x ^ y) & -(x < y));
}

int findMax(int x, int y) {
    return x ^ ((x ^ y) & -(x < y));
}
```

5. Understanding OverFlow through Bits
In computer systems, overflow occurs when the result of an arithmetic operation is too large to be represented within the available number of bits. Let's consider a 3-bit system, which means we have three bits available to represent numbers

Binary	Decimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

```
Now let's perform an addition operation with the numbers 7 and 2:

   111 (7)
 + 010 (2)
 --------
 1001 (9)

 The sum is 1001, which is 9 in decimal. However, in our 3-bit system, we can only
 represent numbers up to 8.
 So in this case, overflow has occurred because the result exceeds
 the maximum representable value.
```

Overflow detection works by examining the carry-out of the leftmost bit, similar to the previous example. If the carry-out is different from the carry-in, overflow has occurred

Decimal	Binary (4-bit)	Binary (3-bit)
8	1000	000
9	1001	001
10	1010	010
11	1011	011
12	1100	100
13	1101	101
14	1110	110
15	1111	111

Please note that the overflow occurs when trying to represent numbers that exceed the maximum value that can be represented in the target bit system. In this case, the 3-bit system can represent numbers from 0 to 7, so any number greater than 7 will cause an overflow when squeezed into 3 bits.

To handle overflow, we would need to widen the number of bits or use alternative representations to accommodate larger values.

6. Some properties of bitwise operations:

a. Addition:

```
1.  a+b = a|b + a&b
```

Now this is a very basic thing.
But how did we get this? Suppose we have two binary numbers 1010 and 0101, there is no chance of any carry in binary addition.In that case we can write :
a+b =a|b
But when we have carry, suppose we have : 1101(a) and 0101(b) then a & b works as the carry which we add further and the equation turns into :

a+b=a|b + a&b

```
2. a+b=a⊕b+2(a&b)
```

It comes from the first equation that I described.But now let's break a & b here and bring xor into action:

We can express a | b as a⊕b + a&b which brings the equation :

a+b=a⊕b+2(a&b)

```
Here  is above explained Addition props:
1. (a|b) = (a+b) - (a&b)
 This is helpful when we want to related AND/OR operations with sum.

2. (a+b) = (a^b) + 2*(a&b)
 This one is a very special relation which can be used to solve some seemingly
 very tough questions.
```

b. Subtraction:

```
a-b = (a⊕(a&b))-((a|b)⊕a)
a-b = ((a|b)⊕b)-((a|b)⊕a)
a-b = (a⊕(a&b))-(b⊕(a&b))
a-b = ((a|b)⊕b)-(b⊕(a&b))
```

c. More Operations

```
a|b = a⊕b + a&b
a⊕(a&b) = (a|b)⊕b
b⊕(a&b) = (a|b)⊕a
(a&b)⊕(a|b) = a⊕b
```

My Other post:
Topics-which-you-cant-skip-interview-preparation-part-1
Topics-which-you-cant-skip-interview-preparation-part-2
Sliding Window
Binary Search

I hope you enjoyed this article! If you found it helpful, please upvote and share it with your friends.
BUY ME A Coffee

bit-manipulation | bit | bit masking | bit compression | bit-swap | bitconcept

Comments: 7 — Best | Most Votes | Newest to Oldest | Oldest to Newest

Type comment here... (Markdown is supported)

Post

imanishkumar7545 ★ 3104 Last Edit: 2 days ago

This is the second time I'm posting this, hoping it reaches a broader audience. Apologies for the repost I haven't posted any articles recently, but now I'm planning to share more.
Can you suggest some topics?
You Can **BUY ME A Coffee through Upi ** -> **manishk3025@fifederal**

15 Hide 5 replies Reply

yshrini ★ 537 a day ago
Graph patterns
0 Reply

123GJWEQ2 ★ 468 a day ago
linear search and permutation tree
0 Reply

its_rohit_3710 ★ 2 2 days ago
All Recursion and DP patterns.
2 Reply

eugcomax ★ 26 2 days ago
FFT

0 Reply

Bishwash_Kumar_Sah ★ 87 2 days ago

Binary Search

2 Reply

Amit_Dahiya_08 ★ 2 a day ago

Can you upload the good question list for bit Manipulation

2 Reply

imanishkumar7545 ★ 3104 Last Edit: 2 days ago

Hye guys I have made youtube channel if I got 1000 subscriber then I will post detail videos on each articles so do subscribe
Visit YouTube Channel

2 Reply

BROCODES2024 ★ 12 an hour ago

Thanks

0 Reply

Parthiv_patel015 ★ 1 a day ago

Great Post !!!!!!!!!

0 Reply

rituSingh03 ★ 1 a day ago

This is so helpful. So glad I came across this! I've never understood bit manipulation well :)

0 Reply

amanjaiswal7236 ★ 0 a day ago

Nice Work

0 Reply