**BitMask**

**1.Subsets**

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

**Note:** The solution set must not contain duplicate subsets.

**Example:**

**Input:** nums = [1,2,3]

**Output:**

[

  [3],

  [1],

  [2],

  [1,2,3],

  [1,3],

  [2,3],

  [1,2],

  []

]

**Solution: C++**

**Approach Bitmask:**

class Solution {

public:

    vector<vector<int>> subsets(vector<int>& nums) {

    int n=nums.size();

         int comb=1<<n;

       vector<vector<int>> output;

        for(int mask=0;mask<comb;mask++){

            vector<int> v;

            for(int j=0;j<n;j++){

                if(mask&(1<<j)){

                    v.push\_back(nums[j]);

                }

            }

            output.push\_back(v);

        }

        return output;

    }

};

**Approach:**

class Solution {

public:

    vector<vector<int>> subsets(vector<int>& nums) {

      vector<vector<int>> v;

   int n=nums.size();

       for(int i=0;i<n;i++){

      vector<vector<int>> temp(v);

           for(int k=0;k<temp.size();k++){

               vector<int> newval(temp[k]);

                   newval.push\_back((nums[i]));

               v.push\_back(newval);

           }

              vector<int> alone;

           alone.push\_back(nums[i]);

           v.push\_back(alone);

       }

         vector<int> p;

        v.push\_back(p);

        return v;

    }

};

**JAVA:**

 public List<List<Integer>> subsets(int[] nums) {

          int n=nums.length;

         int comb=1<<n;

       List<List<Integer>> output=new ArrayList();

        for(int mask=0;mask<comb;mask++){

          List<Integer> curr=new ArrayList();

            for(int j=0;j<n;j++){

                if((mask & (1<<j))!=0){

                    curr.add(nums[j]);

                }

            }

            output.add(curr);

        }

        return output;

    }

1. **Single Number**

Given a **non-empty** array of integers, every element appears *twice* except for one. Find that single one.

**Note:**

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

**Example 1:**

**Input:** [2,2,1]

**Output:** 1

**Example 2:**

**Input:** [4,1,2,1,2]

**Output:** 4

**Solution:**

**C++:**

  int xorv=nums[0];

        for(int i=1;i<nums.size();i++){

            xorv=xorv^nums[i];

        }

        return xorv;

    }

**JAVA**

class Solution {

    public int singleNumber(int[] nums) {

        int xor=nums[0];

        if(nums.length==1){

            return xor;

        }

        for(int i=1;i<nums.length;i++){

            xor=xor^nums[i];

        }

        return xor;

    }

}

1. **Majority Element**

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

**Example 1:**

**Input:** [3,2,3]

**Output:** 3

**Example 2:**

**Input:** [2,2,1,1,1,2,2]

**Output:** 2

**Solution JAVA:**

   public int majorityElement(int[] nums) {

       int mcount=0;

        int ele=Integer.MIN\_VALUE;

        for(int i=0;i<nums.length;i++){

            if(mcount==0){

                ele=nums[i];

            }

            mcount+=(ele==nums[i]?1:-1);

        }

        return ele;

    }

}

**Solution C++**

   int majorityElement(vector<int>& nums) {

        int mcount=0;

        int ele=INT\_MIN;

        for(int i=0;i<nums.size();i++){

            if(mcount==0){

                ele=nums[i];

            }

            mcount+=(ele==nums[i]?1:-1);

        }

        return ele;

    }

1. **Reverse Bits**

Reverse bits of a given 32 bits unsigned integer.

**Example 1:**

**Input:** 00000010100101000001111010011100

**Output:** 00111001011110000010100101000000

**Explanation:**The input binary string **00000010100101000001111010011100** represents the unsigned integer 43261596, so return 964176192 which its binary representation is **00111001011110000010100101000000**.

**Example 2:**

**Input:** 11111111111111111111111111111101

**Output:** 10111111111111111111111111111111

**Explanation:**The input binary string **11111111111111111111111111111101** represents the unsigned integer 4294967293, so return 3221225471 which its binary representation is **10111111111111111111111111111111**.

**Note:**

* Note that in some languages such as Java, there is no unsigned integer type. In this case, both input and output will be given as signed integer type and should not affect your implementation, as the internal binary representation of the integer is the same whether it is signed or unsigned.
* In Java, the compiler represents the signed integers using [2's complement notation](https://en.wikipedia.org/wiki/Two%27s_complement). Therefore, in **Example 2** above the input represents the signed integer -3 and the output represents the signed integer -1073741825.

**Solution:**

**JAVA:**

public class Solution {

    // you need treat n as an unsigned value

    public int reverseBits(int n) {

          int ans=0;

       BitSet bits = new BitSet(32);

        int i=0;

        int j=31;

        while(i<j){

            int first=(n & (1<<i));

            int second=(n & (1<<(j)));

            if(second!=0){

                bits.set(i,true);

            } else{

                 bits.set(i,false);

            }

            if(first!=0 ){

                bits.set(j,true);

            }else{

                bits.set(j,false);

            }

            i++;

            j--;

    }

       int intValue = 0;

        for (int bit = 0; bit < bits.length(); bit++) {

            if (bits.get(bit)) {

                intValue |= (1 << bit);

            }

        }

        return intValue;

    }

}

**C++:**

    uint32\_t reverseBits(uint32\_t n) {

        uint32\_t ans=0;

        bitset<32> bits;

        int i=0;

        int j=31;

        while(i<j){

            int first=(n & (1<<i));

            int second=(n & (1<<(j)));

            bits[i]=second;

            bits[j]=first;

                    i++;

            j--;

    }

      ans=(uint32\_t)bits.to\_ullong();

        return ans;

    }

1. **Number of 1 Bits**

Write a function that takes an unsigned integer and return the number of '1' bits it has (also known as the [Hamming weight](http://en.wikipedia.org/wiki/Hamming_weight)).

**Example 1:**

**Input:** 00000000000000000000000000001011

**Output:** 3

**Explanation:**The input binary string **00000000000000000000000000001011** has a total of three '1' bits.

**Example 2:**

**Input:** 00000000000000000000000010000000

**Output:** 1

**Explanation:**The input binary string **00000000000000000000000010000000** has a total of one '1' bit.

**Example 3:**

**Input:** 11111111111111111111111111111101

**Output:** 31

**Explanation:**The input binary string **11111111111111111111111111111101** has a total of thirty one '1' bits.

**Solution: JAVA**

public class Solution {

    // you need to treat n as an unsigned value

    public int hammingWeight(int n) {

        int count=0;

        for(int i=0;i<32;i++){

            if((n & (1<<i))!=0){

                count++;

            }

        }

        return count;

    }

}

**Solution : C++**

    int hammingWeight(uint32\_t n) {

        int count=0;

        for(int i=0;i<32;i++){

            if((n & (1<<i))!=0){

                count++;

            }

        }

        return count;

    }

};

1. **Missing Number**

Given an array containing *n* distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

**Example 1:**

**Input:** [3,0,1]

**Output:** 2

**Example 2:**

**Input:** [9,6,4,2,3,5,7,0,1]

**Output:** 8

**Note**:  
Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

Solution C++:

class Solution {

public:

    int missingNumber(vector<int>& nums) {

        int xorv=nums[0];

        for(int i=1;i<nums.size();i++){

            xorv^=nums[i];

        }

        for(int i=0;i<=nums.size();i++){

            xorv^=i;

        }

        return xorv;

    }

};

Solution JAVA:

  public int missingNumber(int[] nums) {

          int xorv=nums[0];

        for(int i=1;i<nums.length;i++){

            xorv^=nums[i];

        }

        for(int i=0;i<=nums.length;i++){

            xorv^=i;

        }

        return xorv;

    }

1. **Sum of Two Integer**

Calculate the sum of two integers *a* and *b*, but you are **not allowed** to use the operator + and -.

**Example 1:**

**Input:**a = 1, b = 2

**Output:**3

**Example 2:**

**Input:**a = -2, b = 3

**Output:**1

**Solution:**

    public int getSum(int a, int b) {

        while(b!=0){

            int carry=a&b;

            a=a^b;

            b=carry<<1;

        }

        return a;

    }

1. **Bitwise And of range of numbers**

**Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.**

**Example 1:**

**Input: [5,7]**

**Output: 4**

**Example 2:**

**Input: [0,1]**

**Output: 0**

**Solution:**

class Solution {

    public int rangeBitwiseAnd(int m, int n) {

        int ans=0;

        for(int i=30;i>=0;i--){

            if((n&(1<<i))!=(m&(1<<i))){

                break;

            }else{

                ans|=(m&(1<<i));

            }

        }

        return ans;

    }

}

1. **Single Number II**

**Given a non-empty array of integers, every element appears *three* times except for one, which appears exactly once. Find that single one.**

**Note:**

**Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?**

**Example 1:**

**Input: [2,2,3,2]**

**Output: 3**

**Example 2:**

**Input: [0,1,0,1,0,1,99]**

**Output: 99**

**Solution:**

class Solution {

    public int singleNumber(int[] nums) {

        int arr[]=new int[32];

        for(int i=0;i<32;i++){

            for(int j=0;j<nums.length;j++){

                if((nums[j] & (1<<i))!=0){

                    arr[i]++;

                    arr[i]=arr[i]%3;

                }

            }

        }

        int ans=0;

        for(int i=0;i<32;i++){

            ans|=(arr[i]<<i);

        }

        return ans;

    }

}

**10. Counting Bits**

**Given a non negative integer number num. For every numbers i in the range 0 ≤ i ≤ num calculate the number of 1's in their binary representation and return them as an array.**

**Example 1:**

**Input: 2**

**Output: [0,1,1]**

**Example 2:**

**Input: 5**

**Output: [0,1,1,2,1,2]**

**Follow up:**

* **It is very easy to come up with a solution with run time O(n\*sizeof(integer)). But can you do it in linear time O(n) /possibly in a single pass?**
* **Space complexity should be O(n).**
* **Can you do it like a boss? Do it without using any builtin function like \_\_builtin\_popcount in c++ or in any other language.**

**Solution:**

class Solution {

    public int[] countBits(int num) {

        int countbits[]=new int[num+1];

        countbits[0]=0;

        if(num==0){

            return countbits;

        }

        countbits[1]=1;

        if(num==1){

            return countbits;

        }

        int curcounter=2;

        int oldcounter=0;

        for(int i=2;i<=num;i++){

            if(curcounter==i){

                oldcounter=curcounter;

                curcounter\*=2;

                countbits[i]=1;

            }else if(i%2==0){

                countbits[i]=countbits[oldcounter]+countbits[i-oldcounter]; //even

            }else{

                countbits[i]=countbits[i-1]+1; //odd

            }

        }

        return countbits;

    }

}

**10. Majority Element**

**Given an array of size *n*, find the majority element. The majority element is the element that appears more than**⌊**n/2**⌋**times.**

**You may assume that the array is non-empty and the majority element always exist in the array.**

**Example 1:**

**Input: [3,2,3]**

**Output: 3**

**Example 2:**

**Input: [2,2,1,1,1,2,2]**

**Output: 2**

**Solution:**

class Solution {

    public int majorityElement(int[] nums) {

       int mcount=0;

        int ele=Integer.MIN\_VALUE;

        for(int i=0;i<nums.length;i++){

            if(mcount==0){

                ele=nums[i];

            }

            mcount+=(ele==nums[i]?1:-1);

        }

        return ele;

    }

}