Bitmasking & Bit Manipulation

Karun Karthik

// unique numbers — where every no. occurs 2 times except 1
to find unique no. do XOR sum of all nos.

// odd_check.

```
bool isOdd (int n) {
    return n & 1;
}
```

if n&1.
1 then odd
0 then even

// to get i^{th} bit. from right.

N = 5
i = 2.

```
  2 1 0
  ← 
  0 0 0 1 0 1
      =
```

to mask by ith bit use 1 << i

=> (N & (1 << i)) > 0.

→ true ; 1 bit
→ false ; 0 bit.

```
int getBit (int n, int i) {
    int mask = (1 << i);
    int bit = (n&mask > 0) ? 1 : 0;
    return bit;
}
```

// Set i^{th} bit

```
int setbit(int &n, int i) {
    int mask = (1 << i);
    int ans = n | mask;
    return ans;
}
```

// clear bit.

```
void clearBit(int &n, int i) {
    int mask = ~(1 << i);
    n = n & mask;
}
```

// vpdati a bit

```
void updatiBit(int &n, int i, int v) {
    int mask = ~(1 << i);
    int cleared_n = n & mask;
    n = cleared_n | (v << i);
}
```

// clear last i bits.

```
int clearLastIBits(int n, int i) {
    int mask = (-1 << i);
    return n & mask;
}
```

Clear a range of bits (i to j) — index

```
int clearRangeItoJ (int n, int i, int j) {
    int ones = (~0);
    int a = ones << (j+1);
    int b = (1<<i) - 1;
    int mask = a | b;
    int ans = n & mask;
    return ans;
}
```

⇒ 11100011

a 11100000

b 00000011

mask 11100011

// Replace bits in N by M from i to j.

9 8 7 6 5 4 3 2 1 0

Eg N = 1000 000 0000.

M = 10101 i = 2 j = 6

O/P → N = 10001010100.

i) clear N from i to j (use fn above)

ii) left shift M by i places.

iii) then do or operation

```
int replaceBits (int n, int m, int i, int j) {
    int n_ = clearRangeItoJ(n, i, j);
    int ans = n_ | (m<<i);
    return ans;
}
```

// count number of set bit.

Given n = 13, binary of 13 = 1101 ∴ O/P = 3.

3 methods

① to get last bit use (N & 1) then add 1 to count

then to go to next bit use N = N >> 1

(i.e discard right bit)

& continue till N > 0.

```
{   int ans = 0;
    while (n > 0) {

        ans = ans + (n & 1);

        n = n >> 1;

    }
    return ans;
```

complexity => O(log N + 1)

long long int → 10^{18}.

---

②
```
ans = 0;
while (n > 0) {
    n = n & (n-1);
    ans++;
}
```

// Ans is no. of time while loop runs.

n = n & (n-1)

↳ removes bits from right to left

n = 9.

n = 1001 ans = 1
n-1 = 1000
1000

n = 1000 ans = 2
n-1 = 0111
0000

=> 2 bits

Complexity → O(no. of set bits)

worst case → O(log n).

③ __builtin_popcount(n) → directly gives you the no. of set bits.

---

decimal to binary

```
int decimalToBinary(int n) {
    int ans = 0;
    int p = 1;
    while (n > 0) {
        int last_bit = (n & 1);
        ans = ans + (p * last_bit);
        p = p * 10;
        n = n >> 1;
    }
    return ans;
}
```

n = 13. (④③②①) 1101.

① last bit = 1
ans = 0 + 1*1 = 1.
p = 10.

② last bit = 0.
ans = 1 + 0
p = 100.

③ last bit = 1.
ans = 101
p = 1000

④ last-bit = 1.
ans = 1101.
p = 10000

⑤ n = 0.

---

unique no - II → find unique no in O(n) & no extra space.

eg a = [1,1,2,3,5,2,5,7] o/p ⇒ 3,7.

① XOR all array, then result = XOR.
↳ can never be zero
⇒ the no. will contain atleast 1 set bit.

eg 5,1,2,1,2,3,5,7

result = 5 ^ 1 ^ 2 ^ 1 ^ 2 ^ 3 ^ 5 ^ 7

= 3 ^ 7 = 0111
0011
0100 → 4. ⇒ at 2nd pos set bit is present
3210

→ then we make a list of element which has set bit at second pos.

⇒ a = {5 ^ 5 ^ 7} = 7.

now for second unique no b = result ^ a
= 3 ^ 7 ^ 7 = 3

① to find XOR → res = res ^ no.

② to find unique set bit → while((temp&1) != 1){
pos++;
temp = temp>>1;

③ unique out of 3 (A)

eg. 7
1 1 1 2 2 2 3

O/P → 3.

⟹ write binary rep.

* if we consider each bit pos, sum of all of them will be 3N or 3N+1.

eg: 3, 3, 2, 1, 1, 1, 3.

3 → 0 1 1
3 → 0 1 1
2 → 0 1 0
1 → 0 0 1
1 → 0 0 1
1 → 0 0 1
3 → 0 1 1

0, 4, 6

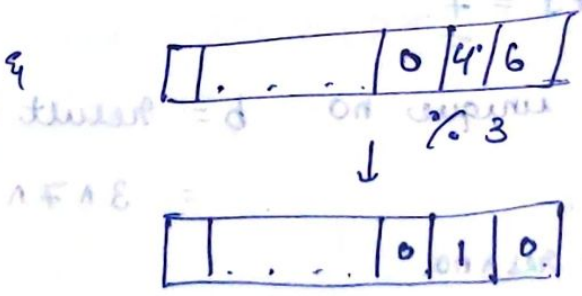
↓ ↓ ↳ 3N

3N 3N+1

largest int = 64 bits

. so store sum in this array of

arr[64];



↳ then convert to decimal

⇒ 2 → uniqueNo

```
int main() {

    int cnt[64] = {0};
    int n, no;
    cin >> n;
    for (int i=0; i<n; i++) {

        cin >> no;
        //update cnt array by extracting bits.
        int j = 0;
        while (no > 0) {
            int last_bit = (no & 1);
            cnt[j] = cnt[j] + last_bit;
            j++;
            no = no >> 1;
        }

    }
    //iterate our array & convert it to no.
    int p = 1;
    int ans = 0;
    for (int i=0; i<64; i++) {
        cnt[i] %= 3;
        ans += (cnt[i] * p);
        p = p << 1;

    }
    cout << ans;

    return 0;
}
```

Fast exponentiation

$a^5 = a^{101} = a^{1\,0\,1}$ (powers $2^2\ 2^1\ 2^0$) $= a^{4+0+1}$

$= a^4 \cdot a^0 \cdot a^1$

$= a^5$.

When we have N then iterating takes $\log_2 N$

Eg $a = 3$, $N = 5$. initial ans = 1.

$a^5 = a^{101}$ (k)

~~ans = 1.~~

$a = 3^1$ k = 1. then ans = 3. 1

$a = 3^2$ k = 0. then ans = 3.

$a = (3^1)^2$ k = 1. then ans = $3 * 3^4$

$= 3^4$.

$= 3^5 = 243$.

Code.

```
int power_optimised (int a, int n) {
    int ans = 1;
    while (n>0) {
        int last_bit = (n&1);
        if(last_bit) {          // if set. then multiply
            ans = ans * a;
        }
        a = a * a;              // square the val.
        n = n >> 1;             // Discard last bit.
    }
    return ans;
}
```

Find the subsets of given string

i/p → abc

o/p → "", a, ab, abc, ac, b, bc, c

Approach →

Subsequences

a b c → every character can be part of result / not.

2 × 2 × 2

↳ 8 possible subsets.

2 components. → iterating over strings. ①

extracting correct elements ②

from 0 to $2^N - 1$ ← [N is length of string]

using a function → filter

f(N, "abc")

↓ ↓

101 ac (a _ c)

$2^N - 1$

↳ $(1 << N) - 1$

```
void printSubsets(char a[]){
    int n = strlen(a);
    for(int i=0; i<(1<<n); i++)
    {
        filterChars(i, a);
    }
}
```

```
void filterChars(int n,
        char a[]){
    int j=0;
    while(n>0){
        int last_bit = (n&1);
        if(last_bit==1){
            cout<<a[j];
        }
        j++;
        n = n>>1;
    }
}
```