আরও

I, ME AND MYSELF !!!

TUESDAY, DECEMBER 22, 2009

Bitwise operations in C: Part 3

Bitwise Operation: Some applications

Back to Part 2

Binary number system is closely related with the powers of 2, and these special numbers always have some amazing bit-wise applications. Along with this, some general aspects will be briefly shown here.

Is a number a power of 2?
How can we check this? of course write a loop and check by repeated division of 2. But with a simple bit-wise operation, this can be done fairly easy.
We, know, the binary representation of p = 2^n is a bit string which has only 1 on the n^{th} position (0 based indexing, right most bit is LSB). And p-1 is a binary number which has 1 on 0 to n-1 th position and all the rest more significant bits are 0. So, by AND-ing p and (p-1) will result 0 in this case:

```
p   = ....01000 &rArr 8
p-1 = ....00111 &rArr 7
-----------------------
AND = ....00000 &rArr 0
```

No other number will show this result, like AND-ing 6 and 5 will never be 0.

Swap two integers without using any third variable:
Well, as no 3rd variable is allowed, we must find another way to preserve the values, how about we some how combine two values on one variable and the other will then be used as the temporary one...

```
Let A = 5 and B = 6
A = A ^ B = 3 /* 101 XOR 110 = 011 */
B = A ^ B = 5 /* 011 XOR 110 = 101 */
A = A ^ B = 6 /* 011 XOR 101 = 110 */
So, A = 6 and B = 5
```

Cool!!!

Divisibility by power of 2
Use of % operation is very slow, also the * and / operations. But in case of the second operand is a power of 2, we can take the advantage of bit-wise operations.
Here are some equivalent operations:

Here, P is in the form 2^X and N is any integer (typically unsigned)

```
N % P = N & (P-1)
N / P = N >> X
N * P = N << X
```

A lot faster and smarter...

Followers (537)

Follow

SUBSCRIBE

Posts

Comments

BLOG HITS

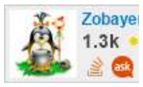
BLOG ARCHIV

- ► 2015 (4)
- ► 2014 (6)
- ► 2013 (19)
- ► 2012 (14)
- ► 2011 (15)
- ► 2010 (33)
- ▼ 2009 (27)
  - ▼ December
    - Happy New
    - Dijkstra's A
    - Linked List
    - Bitwise ope
    - Bitwise ope
    - Bitwise ope
    - Power Seri
    - Gauss–Jor
    - Attacking R
    - Drawing Re
    - Practice Re
  - ► November
  - ► October (1
  - ► Septembe
  - ► August (1)
  - ► July (8)

ABOUT ME

About the % operation : the above is possible only when P is a power of 2

STACK OVERF



Masking operation
What is a mask? Its a way that is used to reshape something. In bit-wise operation, a masking operation means to reshape the bit pattern of some variable with the help of a bit-mask using some sort of bit-wise operation. Some examples following here (we won't talk about actual values here, rather we'll look through the binary representation and using only 16 bits for our ease of understanding):

Grab a portion of bit string from an integer variable.
Suppose A has some value like A = ... 0100 1101 1010 1001
We need the number that is formed by the bit-string of A from 3rd to 9th position.

[Lets assume, we have positions 0 to 15, and 0th position is the LSB]
Obviously the result is B = ... 01 1010 1; [we simply cut the 3rd to 9th position of A by hand]. But how to do this in programming.

Lets assume we have a mask X which contains necessary bit pattern that will help us to cut the desired portion. So, lets have a look how this has to be done:

```
A = 0100 1101 1010 1001
X = 0000 0011 1111 1000
```

See, we have X which have 1s in 3rd to 9th position and all the other thing is 0. We know, AND-ing any bit b with 1 leaves b unchanged.

```
So, X = X & A
Now, we have,
X = 0000 0001 1010 1000
```

So, now we've cut the desired portion, but this is exactly not what we wanted. We have 3 extra 0s in the tail, So just right-shift 3 times:

```
X = X >> 3 = 0000 0000 0011 0101; // hurrah we've got it.
```

Well, I got everything fine, but how to get such a weird X?

```
int x = 0, i, p=9-3+1; // p is the number of 1s we need
for(i=0; i<p, i++)
    x = (x << 1) | 1;
x = x << 3; // as we need to align its lsb with the 3rd bit of A



Execution:
X = 0000 0000 0000 0000 (initially X=0)
X = 0000 0000 0000 0001 (begin loop i=0)
X = 0000 0000 0000 0011 (i=1)
X = 0000 0000 0000 0111 (i=2)
X = 0000 0000 0000 1111 (i=3)
X = 0000 0000 0001 1111 (i=4)
X = 0000 0000 0011 1111 (i=5)
X = 0000 0000 0111 1111 (i=6 loop ends)
X = 0000 0011 1111 1000 (X=X<<3)
```

So, following the same approach, we may invert some portion of a bit-string (using XOR), make all bits 1 (using OR), make all bits in the portion 0 (using AND) etc etc very easily...

So, these are some tasks for the learner,
You have an integer A with some value, print the integers which have:
1. same bit pattern as A except it has all bits 0 within 4th to 23rd position.
2. same bit pattern as A except it has all bits 1 within 9th and 30th position.
3. same bit pattern as A except it has all bits inverted within positions 2nd and 20th.
4. totally inverted bit pattern.

Subset pattern:
Binary numbers can be used to represent subset ordering and all possible combination of taking n items.

For example, a problem might ask you to determine the n'th value of a series when sorted, where each term is some power of 5 or sum of some powers of 5.
It is clear that, each bit in a binary representation correspondence to a specific power of two in increasing order from right to left. And if we write down the consecutive binary values, we get some sorted integers. Like:

```
3 2 1 0 ⇒ power of 2
0 0 0 0 = 0 // took no one
0 0 0 1 = 1 // took power 0
0 0 1 0 = 2 // took power 1
0 0 1 1 = 3 // took power 1 and 0
0 1 0 0 = 4 // took power 2
0 1 0 1 = 5 // took power 2 and 0
0 1 1 0 = 6 // took power 2 and 1
0 1 1 1 = 7 // took power 2, 1 and 0
...........
...........
...........
```

So, what we do here is, take a power of 2 or take the sum of some powers of 2 to get a sorted sequence. Well, if this work for 2.. this will also work for 5 in the same way... Instead of taking the power of 2, we'll take the power of 5.
So the n'th term will be the sum of those powers of 5 on which position n's binary representation has 1. So the 10th term is $5^3 + 5^1$; As, $10_{10} = 1010_2$, it has 1 in 3rd and 1st position.

Worse than worthless
A bitwise GCD algorithm (wikipedia), translated into C from its assembly routine.

Sieve of Eratosthenes (SOE)
This is the idea of compressing the space for flag variables. For example, when we generate prime table using SOE, we normally use 1 int / bool for 1 flag, so if we need to store 10^8 flags, we barely need 100MB of memory which is surely not available... and using such amount of memory will slow down the process. So instead of using 1 int for 1 flag, why don't we use 1 int for 32 flags on each of its 32 bits? This will reduce the memory by 1/32 which is less than 4MB :D

```
#define MAX 100000000
#define LMT 10000

unsigned flag[MAX>>6];

#define ifc(n) (flag[n>>6]&(1<<((n>>1)&31)))
#define isc(n) (flag[n>>6]|=(1<<((n>>1)&31)))

void sieve() {
    unsigned i, j, k;
    for(i=3; i<LMT; i+=2)
        if(!ifc(i))
            for(j=i*i, k=i<<1; j<MAX; j+=k)
                isc(j);
}
```

ifc(x) checks if x is a composite (if correspondent bit is 1)
isc(x) sets x as a composite (by making correspondent bit 1)

Other than this famous bit-wise sieve, we could use the technique in many places, to reduce memory for flagging.

Keep going on...

Posted by Zobayer Hasan at 1:36 AM

17 comments:

SHAON February 4, 2011 at 1:51 PM