位运算

程序中的所有数在计算机内存中都是以二进制的形式储存的。

位运算是对整数在内存中的二进制位进行操作。

由于位运算直接对内存数据进行操作，不需要转成十进制，因此处理速度非常快。

各种位运算

1.与运算

C++符号：&

运算规则：运算数为0和1时，除了1&1=1，其余情况均为0

简单应用：常用来判断一个整数的奇偶性

2.或运算

C++符号：|

运算规则：运算数为0和1时，除了0|0=0，其余情况均为1

简单应用：将一个数变为不超过它的最接近偶数（x|1-1）

3.异或运算

C++符号：^

运算规则：运算数为0和1时，除了0|0=0，其余情况均为1

简单应用：通常用于对二进制的特定一位进行取反操作

异或的逆运算是自己本身：(a ^ b) ^ b = a

利用这一点：可以用异或实现两数交换

a=a^b,b=a^b,a=a^b

4.取反

C++符号：~　（注意不是-）

辨析：lowbit函数是 x&(~x)还是x&(-x)？

运算规则：把内存中的0和1全部取反

注意有符号整数的操作：~x实际上等于-a-1（负数用补码表示）

例如：~5 等于-6

考虑一个8位有符号整数，5可以表示为：

0000 0101

取反：

1111 1010

高位取反之后说明正数变为负数：

以补码表示（取反+1）

1000 0110

5.左移运算

C++符号：<<

运算规则：把二进制位整体向左移动

简单应用：左移n位相当于乘以2的n次方

6.右移运算

C++符号：>>

运算规则：把二进制位整体向右移动

简单应用：右移n位相当于除以2的n次方

运算符的优先级（从高到低）

优先级	描述	运算符
1	括号	()、[]
2	正负号	+、-
3	自增自减，非	++、--、!
4	乘除，取余	*、/、%
5	加减	+、-
6	移位运算	<<、>>、>>>
7	大小关系	>、>=、<、<=
8	相等关系	==、!=
9	按位与	&
10	按位异或	^
11	按位或	|
12	逻辑与	&&
13	逻辑或	||
14	条件运算	?:
15	赋值运算	=、+=、-=、*=、/=、%=
16	位赋值运算	&=、|=、<<=、>>=、>>>=

功能	位运算
去掉最后一位	x >> 1
在最后加一个0	x << 1
在最后加一个1	x >> 1
把最后一位变成1	x | 1
把最后一位变成0	x | 1 - 1
最后一位取反	x ^ 1
把右数第k位变成1	x | (1 << (k - 1))
把右数第k位变成0	x & ~(1 << (k - 1))
右数第k位取反	x ^ ~(1 << (k - 1))

功能	位运算
取末三位	x & 7
取末k位	x & (1 << k - 1)
取右数第k位	x >> (k - 1) & 1
把末k位变成1	x | (1 << k - 1)
末k位取反	x ^ (1 << k - 1)
把右边连续的1变成0	x & (x+1)
把右起第一个0变成1	x | (x+1)
把右边连续的0变成1	x | (x-1)
取右边连续的1	(x ^ (x + 1)) >> 1
去掉右起第一个1的左边(lowbit)	x & (x ^ (x-1))

位运算优化

- 示例：二进制中的1有奇数个还是偶数个

我们可以用下面的代码来计算一个**32位**整数的二进制中1的个数的奇偶性：

```
int x,c=0;
cin>>x;
for(int i=1;i<=32;i++){
    c+=x&1;
    x>>=1;
}
cout<<(c&1);
```

进一步优化：

```
int x;
cin>>x;
x^=x>>1;
x^=x>>2;
x^=x>>4;
x^=x>>8;
x^=x>>16;
cout<<(x&1);
```

第一次异或得到的二进制数，其右起第i位上的数表示原数中第i和i+1位上有奇数个1还是偶数个1;
第一次异或得到的二进制数，其右起第i位上的数表示原数对应的四个位置有奇数个1还是偶数个1;
。。。
第五次异或后，得到的二进制数的最末位就表示整个32位数里有多少个1。

更多“骚操作” 大家可以看matrix67的原博客《位运算简介及实用技巧》，这里就不再赘述。

例题：求a乘b对p取模的值，其中1<=a,b,p<=10^18。

分析：简单的暴力相乘显然会溢出（相当于对64位整数最大值+1取模），对a进行b次累加显然会超时。

方法1：将b进行二进制拆分，然后与a做乘法分配。

```
typedef long long ll;
inline ll mul(ll a, ll b, ll p) {
    a %= p, b %= p;
    ll ans = 0;
    while (b) {
        if (b & 1) ans = (ans + a) % p;
        a = a * 2 % p, b >>= 1;
    }
    return ans;
}
```

复杂度：O(log2*b*)

还有一个与主题无关的方法（但是复杂度更低）

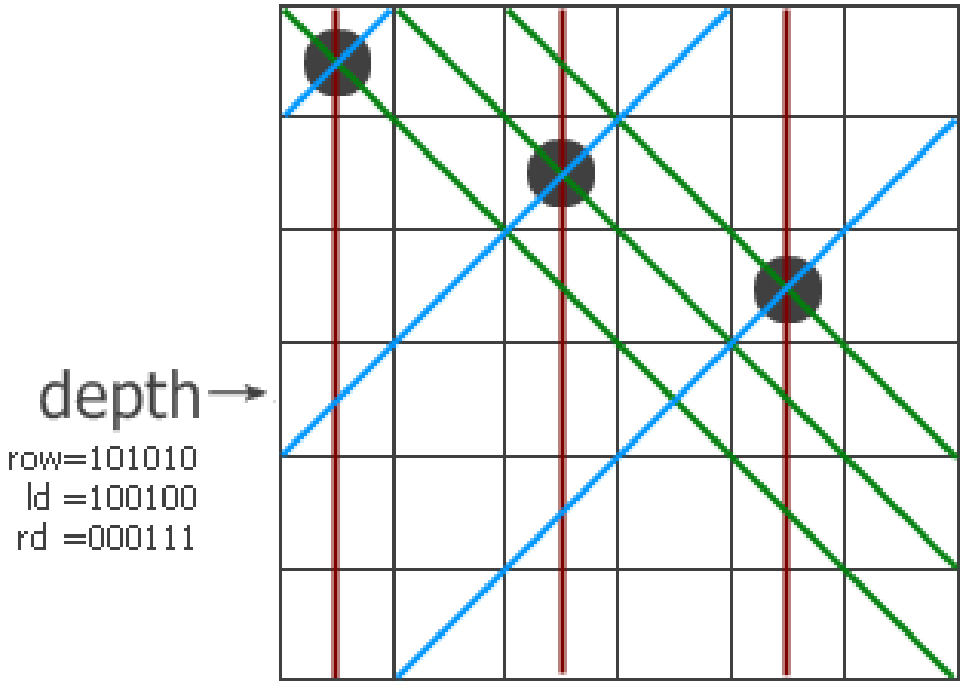
思路：利用long double字节长但是不精确的特点，避开了64位的限制算出答案。

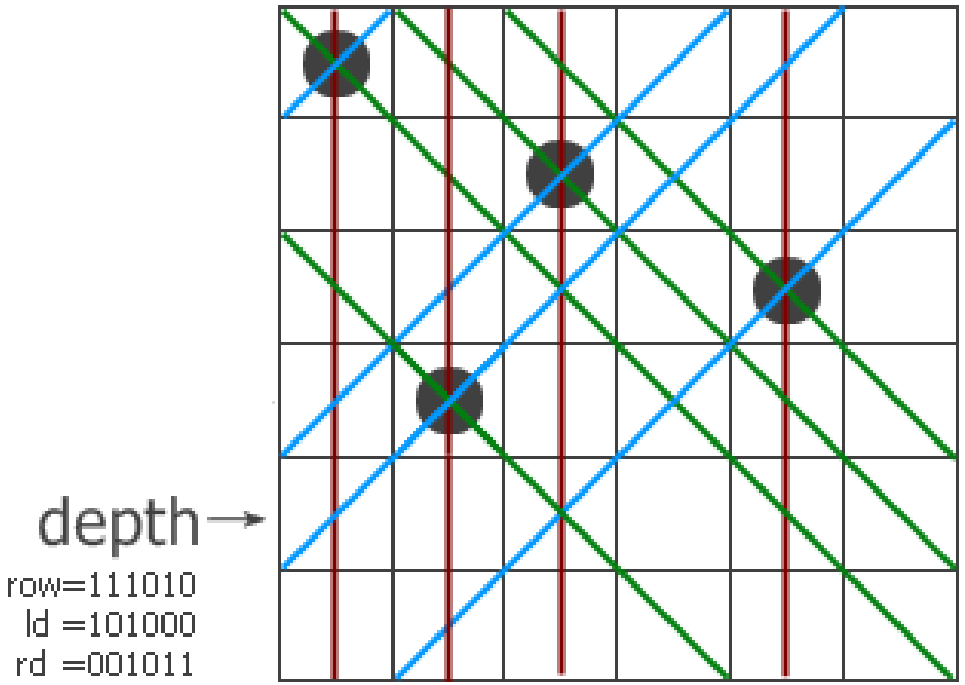
```
inline long long qmul(long long a,long long b,long long p){
    a=(a%p+p)%p;b=(b%p+p)%p;//顺带处理了负数和过大的情况
    long long c=a*(long double)b/p;//这里的c是准确的下取整结果
    long long ans=a*b-c*p;
    //你没看错，这里a*b以及c*p都溢出了，但是差值一定小于p，
    //所以溢出的只有高位，对低位没有影响
    if(ans<0)//如果不在[0,p)之间则调整一下
        ans+=p;
    else if(ans>=p)
        ans-=p;
    return ans;
}
```

位运算与状态压缩

位运算常用来压缩信息：一行的信息可以用一个二进制整数表示

经典案例：n皇后





row、ld和rd，分别表示在纵列和两个对角线方向的限制条件下这一行的哪些地方不能放。它们三个并起来，得到该行所有的禁位，取反后就得到所有可以放的位置（用pos来表示）。以6皇后为例，如果递归到某个时候发现row=111111了，说明六个皇后全放进去了，ans++

```
int dfs(int row,int ld,int rd) {
//row、ld和rd，分别表示在纵列和两个对角线的限制下这一行哪些地方不能放
    int lim=(1<<n)-1; //初始化
    int p,pos;
    if(row!=lim) { //row=lim表示所有n个位置已经放置
        pos=lim&~(row|ld|rd); //pos表示所有可以放的位置
        while(pos) {
            p=pos&-pos; //取出最右边那个1，p就表示该行的某个可以放子的位置
            pos^=p;     //从pos中移除并递归调用
            dfs(row|p,(ld+p)<<1,(rd+p)>>1);
        }
    } else ans++;
}
```

例题：费解的开关 06年noip模拟赛（一） by Matrix67 第四题

25盏灯排成一个5x5的方形。每一个灯都有一个开关，游戏者可以改变它的状态。每一步，游戏者可以改变某一个灯的状态。游戏者改变一个灯的状态会产生连锁反应：和这个灯上下左右相邻的灯也要相应地改变其状态。 我们用数字“1”表示一盏开着的灯，用数字“0”表示关着的灯。下面这种状态

10111 01101 10111 10000 11011	在改变了最左上角的灯的状态后将变成：	01111 11101 10111 10000 11011

给定一些游戏的初始状态（游戏组数<=500），编写程序判断游戏者是否可能在6步以内使所有的灯都变亮。若可以输出具体步数，若不行输出“-1”

一般性解法:

从第一行开始，按行枚举关着的灯，通过开关下一行灯将第一行变为全亮。
若发现最后还有灯是关着的，说明无解否则输出步数。

```
int dfs(int cnt,int k) {
    if(cnt>5) {
        ans=min(ans,check(k));
        return 0;
    }
    a[1][cnt]=!a[1][cnt];
    a[1][cnt-1]=!a[1][cnt-1];
    a[1][cnt+1]=!a[1][cnt+1];
    a[2][cnt]=!a[2][cnt];

    dfs(cnt+1,k+1);//按下这个开关

    a[1][cnt]=!a[1][cnt];
    a[1][cnt-1]=!a[1][cnt-1];
    a[1][cnt+1]=!a[1][cnt+1];
    a[2][cnt]=!a[2][cnt];

    dfs(cnt+1,k);//不按下这个开关
}
```

位运算优化:

```
int change(int x,int y) {
    x^=(1<<y);
    if (y>=5) x^=(1<<(y-5));
    if (y<20) x^=(1<<(y+5));
    if (y%5) x^=(1<<(y-1));
    if (y%5<4) x^=(1<<(y+1));
    return x;
}

void bfs() {
    memset(ans,-1,sizeof(ans));
    int head=0,tail=1;
    q[1]=(1<<25)-1;
    ans[(1<<25)-1]=0;
    while (head<tail) {
        int x=q[++head];
        if (ans[x]==6) return;
        for (int i=0; i<25; i++) {
            int y=change(x,i);
            if (ans[y]<0) q[++tail]=y,ans[y]=ans[x]+1;
        }
    }
}
```

例题2：[SCOI2005]互不侵犯

在N×N的棋盘里面放K个国王，使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共8个格子。

N，K （1 <=N <=9, 0 <= K <= N * N）

将每一行放置国王的状态用一个二进制数记录，
1010可表示第一、三格放国王，二、四格没有放国王

国王与国王之间放置不允许冲突，如何表示？
以国王不允许同列为例：
!(state[a] & state[b])　　state表示对应的二进制数

斜对角怎么办呢？左移或者右移
左上角为例：
!((state[b]<<1) & state[a]))

由于一个国王影响周围8个方向，从上到下考虑需要同时枚举相邻两行的状态：

```
for(int i = 2 ; i <= n ; i++)        //第一行预处理，从第二行枚举
    for(int j = 1; j <= ans ; j++)     //枚举相邻两行的状态；
        for(int p = 1; p <= ans ; p++) {
            if(state[j] & state[p]) continue;  //上下相邻不行
            if(state[j] & (state[p]<<1))    continue;  //右上角不能有国王；
            if((state[j]<<1) & state[p])    continue;  //左上角；
            for(int s = 1 ; s <= k ; s++) {//s表示本行以上用了多少国王；
                if(king[j] + s > k) continue;
                 //本行加以前用过的国王，总数不能超过限定；
                dp[i][j][king[j]+s] += dp[i-1][p][s];
            }
        }
```

实战：

详见XJ19 位运算专题

https://vjudge.net/contest/355650#overview

C题 bitset [Jsoi2016]位运算

题意：你需要在[0,G−1] 中选出n个不同的数,使它们异或起来等于0,问有多少种不同的方案数;G是由一个二进制位小于等于50的二进制数S重复K次得到的

3⩽n⩽7,1⩽k⩽10^5,1⩽|S|⩽50

我们考虑选出的n个数Ai按从大到小排序,那么每次我们取出来的数是这样的:

G>An>An−1>An−2...>A2>A1

这样就能保证我们取的方案是不同的(去除重复情况);

接下来考虑n个二进制长度为|S|的数该怎么取;

我们用一个01序列来表示当前n个数的大小关系。例如10011可表示

G>A5≤A4≤A3>A2>A1;

现在我们考虑逐位得到这n个数的大小关系,已知当前位的大小关系时,只需要再枚举下一位n个数的二进制位,就能推出下一位的大小关系;

用Fij表示当前大小关系状态i取|S|位后到状态j一共有多少种方式,转移时，2^n枚举这一位填什么。然后判断是否合法。我们的目标是求出对于每一种开始的状态，在|S|步后到每种状态的方案数

于是我们便能用如上的状压递推的方式得到这个F;

接下来的考虑取了i个|S|位大小关系是j（第j位状态表示Aj小于或等于Aj+1）时一共有多少种方案,因为每|S|个之间的转移是一样的,取i个|S|那么长实际上就是取F的i次方,能够使用矩阵快速幂快速得到;

复杂度 O(2^(n*3)*|S|+2^(n*3)*logk);

H题 Islands and Bridges POJ2288

题意：

给出n个点，m条边。每个点有一个权值w。找出一条汉密尔顿路径，使它的值最大。一条汉密尔顿路径的值由三部分组成：

1) 路径上每个点的权值之和

2) 路径上每条边u-v，将其权值的积累加起来。即w[u]*w[v]

3) 如果三个点形成一个三角形，例如i、i+1、i+2，那么将

    w[i]*w[i+1]*w[i+2]累加起来

n<=13

思路:

汉密尔顿路径：经过每个点恰好一次的路径。

状压:

dp[p][i][j] p代表当前已经拜访的点的状态，j是前一个点，i是前一个点的前一个点。

转移方程:

```
p[p|1<<k][j][k] = max{dp[p|1<<k][j][k],
                      dp[p][i][j] + w[k] + w[k]*w[j]
                      dp[p][i][j] + w[k] + w[k]*w[j]+w[i]*w[j]+w[k]
                      }
```

核心代码:

```
for(int p = 0; p < (1<<n); p++)  //枚举状态
    for(int i = 0; i < n; i++) if(p & (1<<i))  //取i
        for(int j = 0; j < n; j++) if(p & (1<<j))  //取j
            if(G[i][j] && dp[p][i][j] != -1)
                for(int k = 0; k < n; k++)
                    if(G[j][k] && k != i && !(p & (1<<k))) { //边jk
                        int tmp = dp[p][i][j] + w[k] + w[k] * w[j];
                        if(G[i][k]) //形成三角形
                            tmp += w[i] * w[j] * w[k];
                        if(dp[p|(1<<k)][j][k] < tmp) {
                            dp[p|(1<<k)][j][k] = tmp;
                            ways[p|(1<<k)][j][k] = ways[p][i][j];
                        }//决定是否更改原方案
                        else if(dp[p|(1<<k)][j][k] == tmp) {
                            ways[p|(1<<k)][j][k] += ways[p][i][j];
                        }
                    }
```

bitset

bitset用来存储二进制数位。像一个bool类型的数组一样，bitset每个元素只有0或1两个数值，但是bitset中一个元素一般只占1 bit。

bitset中的每个元素都能单独被访问。

bitset支持所有位运算

Bitset定义及初始化

```cpp
#include<iostream>
#include<bitset> //bitset的头文件
#include<cstring>
using namespace std;
int main() {
    bitset<23>bit (string("11101001"));
    //23表示初始化长度，不足在前补0
    //bitset可以用string和整数（对应二进制）初始化
    cout<<bit<<endl;//00000000000000011101001
    bit=233;
    cout<<bit<<endl;//00000000000000011101001
    return 0;
}
```

Bitset的常用函数

对于一个叫做bit的bitset：
bit.size()　返回大小（位数）
bit.count()　返回1的个数
bit.any()　返回是否有1
bit.none()　返回是否没有1
bit.set()　全都变成1
bit.set(p)　将第p + 1位变成1（bitset是从第0位开始的！）
bit.set(p, x)　将第p + 1位变成x
bit.reset()　全都变成0
bit.reset(p)　将第p + 1位变成0
bit.flip()　全都取反
bit.flip(p)　将第p + 1位取反
bit.to_ulong()　返回它转换为unsigned long的结果，如果超出范围则报错
bit.to_ullong() 返回它转换为unsigned long long的结果，如果超出范围则报错
bit.to_string() 返回它转换为string的结果

前面说到bitset支持位运算

示例：左移运算

```
bitset<23>bit(string("11101001"));
cout<<(bit<<5)<<endl;
//输出00000000000111010010000
```

例题：BZOJ3687：子集的算术和的异或和。

输入：
2
1 3
输出
6
样例解释： 6=1 异或 3 异或 (1+3)

ai >0，1<n<1000，∑ai≤2000000。

思路:
设f[i]表示子集和为i的方案，那么加入一个数x，所有的f[i]+=f[i−1]
考虑到最后的异或操作，因此我们只维护方案的奇偶性即可
这样的话用一个bitset就可以了

代码:

```
int N;
bitset<2000001>bit;
int main() {
    scanf("%d",&N);
    bit[0]=1;
    while(N--) {
        int x;
        scanf("%d",&x);
        bit^=bit<<x;
    }
    int ans=0;
    for(int i=2000000; i>=0; i--)
        if(bit[i]==1)
            ans^=i;
    printf("%d",ans);
```

参考：

http://www.matrix67.com/blog/archives/263

https://www.cnblogs.com/guoguo003/p/3163787.html

https://www.cnblogs.com/zwfymqz/archive/2018/04/02/8696631.html