快速幂运算

```
long long qpower(long long a,long long b,long long mod){
        if(b == 1) return a;
        long long ans=1;
    while(b)
    {
        if(b&1)
            ans=ans*a%mod;
        b>>=1;
        a=a*a%mod;
    }
    return ans;
}
```

求乘法逆元的时候若b为素数直接令b = mod-2 即可
必要的时候使用BigInteger或者__int128防止溢出

欧几里得算法

```
int gcd(int a, int b){
    return b == 0? a: gcd(b,a%b);
}
```

扩展欧几里得算法

```
LL exgcd(LL a,LL b,LL &x,LL&y){
    if(a==0&&b==0) return -1;//无最大公约数
    if(b==0){x=1;y=0;return a;}
    LL d=exgcd(b,a%b,y,x);
    y -= a/b*x;
    return d;
}
```

扩展欧几里得算法求乘法逆元

```
LL mod_reverse(LL a,LL n){
    LL x,y;
    LL d=exgcd(a,n,x,y);
    if(d==1) return (x%n+n)%n;
    else return -1;
}
```

生成素数

```
const int MAXN = 1e4;
int primer[MAXN+1];
void getprimer(){
    memset(primer,0,sizeof(primer));
    for(int i = 2 ; i <= MAXN ; i++){
        if(!primer[i]) primer[++primer[0]] = i;
        for(int j = 1 ; j <= primer[0] && primer[j] <= MAXN / i; j++){
            primer[primer[j]*i] = 1;
            if(i%primer[j] == 0) break;
        }
    }
}
```

因式分解

n 带分解整数 &tot 质因数个数

a 质因数值 b 质因数指数

```
void factor (int n, int a[MAXN],int b[MAXN],int &tot) {
    int temp, i, now;
    temp = (int) ((double) sqrt(n) + 1);
    tot = 0;
    now = n;
    for (i = 2; i <= temp; i++)
        if (now % i == 0) {
            a[++tot] = i;
            b[tot] = 0;
            while (now % i == 0) {
                ++b[tot];
                now /= i;
            }
        }
    if (now != 1) {
        a[++tot] = now;
        b[tot] = 1;
    }
}
```

对于1e14以内的数分解质因数

```cpp
const int MAXN = 1e7+10;
bool notprime[MAXN];//值为false 表示素数，值为true 表示非素数
int p[(int)1e5],e[(int)1e5],factcnt;
void init(){
    memset(notprime,false,sizeof(notprime));
    notprime[0]=notprime[1]=true;
    for(int i=2;i<MAXN;i++)
        if(!notprime[i]){
            if(i>MAXN/i)continue;
            for(int j=i*i;j<MAXN;j+=i)
                notprime[j]=true;
        }
}
void fact(int n){
    int &cnt = factcnt;
    cnt = 0;
    for(int i = 2 ; i < n ; i++){
        if(n%i == 0){
            e[++cnt] = 1;
            p[cnt] = i;
            n/= i;
        }
        while(n % i == 0) {
            n /= i;
            e[cnt]++;
        }
        if(!notprime[n]){
            p[++cnt] = n;
            e[cnt] = 1;
            break;
        }
    }
}
```

欧拉函数

线性欧拉筛

```
void phi_table(int n, int* phi) {
  for (int i = 2; i <= n; i++) phi[i] = 0;
  phi[1] = 1;
  for (int i = 2; i <= n; i++)
    if (!phi[i])
      for (int j = i; j <= n; j += i) {
        if (!phi[j]) phi[j] = j;
        phi[j] = phi[j] / i * (i - 1);
      }
}
```

只求一个数的欧拉函数

```
int euler_phi(int n) {
  int m = int(sqrt(n + 0.5));
  int ans = n;
  for (int i = 2; i <= m; i++)
    if (n % i == 0) {
      ans = ans / i * (i - 1);
      while (n % i == 0) n /= i;
    }
  if (n > 1) ans = ans / n * (n - 1);
  return ans;
}
```

Mobius函数计算

```
void pre() {
  mu[1] = 1;
  for (int i = 2; i <= 1e7; ++i) {
    if (!v[i]) mu[i] = -1, p[++tot] = i;
    for (int j = 1; j <= tot && i <= 1e7 / p[j]; ++j) {
      v[i * p[j]] = 1;
      if (i % p[j] == 0) {
        mu[i * p[j]] = 0;
        break;
      }
      mu[i * p[j]] = -mu[i];
    }
  }
```

进制转换

返回X进制数对应的y进制数

```
string transform(int x , int y , string s){
        string res = "";
        int sum = 0;
        for(int i = 0 ; i < s.length() ; i++){
                if(s[i] == '-') continue;
                if(s[i] >= '0' && s[i] <= '9'){
                        sum = sum * x + s[i] - '0';
                }else{
                        sum = sum * x + s[i] - 'A' + 10;
                }
        }
        while(sum){
                char tmp = sum % y;
                sum /= y;
                if(tmp <= 9){
                        tmp += '0';
                }else {
                        tmp = tmp - 10 +'A';
                }
                res = tmp + res;
        }
        if(res.length() == 0) res = "0";
        if(s[0] == '-') res = '-' + res;
        return res;
}
```

线段树

注意：针对本套模板s = 1，t = n，是整体区间，p=1为根节点，s,t是修改区间

建树

```
void build(int s, int t, int p) {
// 对 [s,t] 区间建立线段树,当前根的编号为 p
        if (s == t) {
                d[p] = a[s];
                return;
        }
        int m = (s + t) / 2;
        build(s, m, p * 2), build(m + 1, t, p * 2 + 1);
        // 递归对左右区间建树
        d[p] = d[p * 2] + d[(p * 2) + 1];
}
```

采用堆式存储（2p是 p的左儿子,2p+1 是p的右儿子），若有 n个叶子结点，则 d 数组的范围最大为 $2^{\lfloor logn \rfloor + 1}$

区间修改（给定区间内每一个元素加值）//如果是乘值，那么直接把代码中的+=替换成*=即可

注意：因为使用了b数组，因此不同操作的查询函数是不一样的，需要自行修改

```
void update(int l, int r, int c, int s, int t, int p) {
  // [l,r] 为修改区间,c 为被修改的元素的变化量,[s,t] 为当前节点包含的区间,p
  // 为当前节点的编号
  if (l <= s && t <= r) {
    d[p] += (t - s + 1) * c, b[p] += c;
    return;
  }  // 当前区间为修改区间的子集时直接修改当前节点的值,然后打标记,结束修改
  int m = (s + t) / 2;
  if (b[p] && s != t) {
    // 如果当前节点的懒标记非空,则更新当前节点两个子节点的值和懒标记值
    d[p * 2] += b[p] * (m - s + 1), d[p * 2 + 1] += b[p] * (t - m);
    b[p * 2] += b[p], b[p * 2 + 1] += b[p];  // 将标记下传给子节点
    b[p] = 0;                                // 清空当前节点的标记
  }
  if (l <= m) update(l, r, c, s, m, p * 2);
  if (r > m) update(l, r, c, m + 1, t, p * 2 + 1);
  d[p] = d[p * 2] + d[p * 2 + 1];
}
int getsum(int l, int r, int s, int t, int p) {
  // [l,r] 为查询区间,[s,t] 为当前节点包含的区间,p 为当前节点的编号
  if (l <= s && t <= r)
    return d[p];  // 当前区间为询问区间的子集时直接返回当前区间的和
  int m = (s + t) / 2, sum = 0;
  if (l <= m) sum += getsum(l, r, s, m, p * 2);
  // 如果左儿子代表的区间 [l,m] 与询问区间有交集,则递归查询左儿子
  if (r > m) sum += getsum(l, r, m + 1, t, p * 2 + 1);
  // 如果右儿子代表的区间 [m+1,r] 与询问区间有交集,则递归查询右儿子
  return sum;
}
```

区间修改（修改值）

```cpp
void update(int l, int r, int c, int s, int t, int p) {
  if (l <= s && t <= r) {
    d[p] = (t - s + 1) * c, b[p] = c;
    return;
  }
  int m = (s + t) / 2;
  if (b[p]) {
    d[p * 2] = b[p] * (m - s + 1), d[p * 2 + 1] = b[p] * (t - m),
          b[p * 2] = b[p * 2 + 1] = b[p];
    b[p] = 0;
  }
  if (l <= m) update(l, r, c, s, m, p * 2);
  if (r > m) update(l, r, c, m + 1, t, p * 2 + 1);
  d[p] = d[p * 2] + d[p * 2 + 1];
}
int getsum(int l, int r, int s, int t, int p) {
  if (l <= s && t <= r) return d[p];
  int m = (s + t) / 2;
  if (b[p]) {
    d[p * 2] = b[p] * (m - s + 1), d[p * 2 + 1] = b[p] * (t - m),
          b[p * 2] = b[p * 2 + 1] = b[p];
    b[p] = 0;
  }
  int sum = 0;
  if (l <= m) sum = getsum(l, r, s, m, p * 2);
  if (r > m) sum += getsum(l, r, m + 1, t, p * 2 + 1);
  return sum;
}
```

取子游戏的SG函数

```
const int N = 1e3+10;
int f[N];//可以取走的石子个数
int SG[N];//0~n的SG函数值
int Hash[N];
void getSG(int n){
    memset(SG,0,sizeof(SG));
    for(int i = 1; i <= n; i++){
        memset(Hash,0,sizeof(Hash));
        for(int j = 1; f[j] <= i; j++)
            Hash[SG[i-f[j]]] = 1;
        for(int j = 0; j <= n; j++){    //求mes{}中未出现的最小的非负整数
            if(Hash[j] == 0){
                SG[i] = j;
                break;
            }
        }
    }
}
```