关于多项式的一点研究及其在ACM竞赛中的应用

1. 多项式的前缀和

1.1 n^k 的前缀和

首先来看几个众所周知的公式

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

用数学归纳法很容易验证上述公式的正确性，但是对于任何给定的非负整数 k ，如何求出 $\sum_{i=1}^{n} i^k$ 呢？下面给出一种利用杨辉三角的计算方法。

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 6 & 10 & 15 \\ 1 & 4 & 10 & 20 & 35 \\ 1 & 5 & 15 & 35 & 70 \end{matrix}$$

杨辉三角的自然数形式

$$\begin{matrix} C_0^0 & C_1^1 & C_2^2 & C_3^3 & C_4^4 \\ C_1^0 & C_2^1 & C_3^2 & C_4^3 & C_5^4 \\ C_2^0 & C_3^1 & C_4^2 & C_5^3 & C_6^4 \\ C_3^0 & C_4^1 & C_5^2 & C_6^3 & C_7^4 \\ C_4^0 & C_5^1 & C_6^2 & C_7^3 & C_8^4 \end{matrix}$$

杨辉三角的组合数形式

不难发现，第i列的前n个数字之和刚好等于第$i+1$列第n个数字，即

$$C_k^k + C_{k+1}^k + \cdots + C_{k+n}^k = C_{k+n+1}^{k+1}$$

证 根据杨辉恒等式 $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ ，

$$\text{左边} = C_k^k + C_{k+1}^k + \cdots + C_{k+n}^k = C_{k+1}^{k+1} + C_{k+1}^k + \cdots + C_{k+n}^k = C_{k+2}^{k+1} + \cdots + C_{k+n}^k = \cdots = C_{k+n+1}^{k+1} = \text{右边}$$

换言之，除第一列外，每一列都是前一列的“前缀和”，而每一列的数字，都是 $C_n^k(k = \text{列数} - 1)$ 的形式，其本质就是 n 的多项式，例如：

$$\text{第一列}: C_n^0 = 1$$

$$\text{第二列}: C_n^1 = \sum_{i=0}^{n-1} C_i^0 = \sum_{i=0}^{n-1} 1 = n$$

$$\text{第三列：} C_n^2 = \sum_{i=1}^{n-1} C_i^1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$\text{第四列：} C_n^3 = \sum_{i=2}^{n-1} C_i^2 = \sum_{i=2}^{n-1} \frac{i(i-1)}{2} = \frac{n(n-1)(n-2)}{6}$$

根据第三列，得到

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n = \frac{n(n+1)}{2}$$

根据第四列，得到

$$\sum_{i=2}^{n-1} \frac{i(i-1)}{2} = \frac{1}{2}\sum_{i=2}^{n-1} i^2 - \frac{1}{2}\sum_{i=2}^{n-1} i = \frac{1}{2}\left(\sum_{i=1}^{n} i^2 - 1 - n^2\right) - \frac{(n+1)(n-2)}{4} = \frac{n(n-1)(n-2)}{6}$$

进而得到

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

至于更高次的求和，以此类推即可，上述计算过程比较机械化，因此不难用计算机来实现

```cpp
#include <bits/stdc++.h>
#define rep(i,a,b) for(ll i=a;i<=b;i++)
using namespace std;
typedef long long ll;
ll gcd(ll a, ll b)
{
    return b ? gcd(b, a%b) : a;
}
class frac
{
    public:
    ll x,y;
    frac(){}frac(ll x,ll y):x(x),y(y){}
    bool operator < (const frac &b)const{return x*b.y<y*b.x;}
    bool operator > (const frac &b)const{return x*b.y>y*b.x;}
    bool operator ==(const frac &b)const{return x*b.y==y*b.x;}
    frac operator + (const frac &b)const{ll d=gcd(x*b.y+b.x*y,y*b.y);return 
frac((x*b.y+b.x*y)/d,(y*b.y)/d);}
    frac operator - (const frac &b)const{ll d=gcd(x*b.y-b.x*y,y*b.y);return 
frac((x*b.y-b.x*y)/d,(y*b.y)/d);}
    frac operator * (const frac &b)const{ll d=gcd(x*b.x,y*b.y);return 
frac((x*b.x)/d,(y*b.y)/d);}
    frac operator / (const frac &b)const{ll d=gcd(x*b.y,b.x*y);return 
frac((x*b.y)/d,(b.x*y)/d);}
    frac operator * (ll b)const{ll d=gcd(x*b,y);return frac((x*b)/d,(y)/d);}
    frac operator / (ll b)const{ll d=gcd(x,y*b);return frac((x)/d,(y*b)/d);}
    frac operator = (ll b){*this=frac(b,1);return *this;}
};
ostream &operator <<(ostream &out,const frac &a)
{
    if(a.y==1)out<<a.x;
    else out<<a.x<<"/"<<a.y;
    return out;
}
```

```
typedef frac type;
bool isZero(type x){
    return x.x==0;
}
class Poly{
public:
    vector<type>a={frac(0,1)};
    Poly(){}
    Poly(vector<type> b):a(b){}
    ll n(){
        return a.size()-1;
    }
    Poly operator = (type b){
        this->a.resize(1);
        this->a[0]=b;
        return *this;
    }
    Poly operator = (vector<type> b){
        this->a=b;
        return *this;
    }
    friend ostream &operator << (ostream &o,const Poly &f){
        for(int i=f.a.size()-1;~i;i--){
            if(!i)cout<<"("<<f.a[i]<<")";
            else cout<<"("<<f.a[i]<<")"<<"x^"<<i<<"+";
        }
        cout<<endl;
    }
    type coef(int i){
        if(i>=a.size() || i<0)return frac(0,1);
        return a[i];
    }
    type& operator [] (int i){
        if(i>=a.size() || i<0)cout<<" Warning: Index out of range\n";
        return a[i];
    }
    type operator () (type x){
        type ans;
        ans=0;
        for(int i=n();~i;i--)ans=ans*x+a[i];
        return ans;
    }
    Poly operator () (Poly x){
        Poly ans,t;
        for(int i=n();~i;i--){
            t=Poly((vector<type>){a[i]});
            ans=ans*x+t;
        }
        return ans;
    }
    Poly operator + (Poly &b){
        Poly c;
        c.a.resize(max(a.size(),b.a.size()));
        for(int i=0;i<c.a.size();i++)c.a[i]=coef(i)+b.coef(i);
        while(c.a.size()>1 && isZero(*(c.a.end()-1)))c.a.erase(c.a.end()-1);
        return c;
    }
    Poly operator - (Poly &b){
```

```
        Poly c;
        c.a.resize(max(a.size(),b.a.size()));
        for(int i=0;i<c.a.size();i++)c.a[i]=coef(i)-b.coef(i);
        while(c.a.size()>1 && isZero(*(c.a.end()-1)))c.a.erase(c.a.end()-1);
        return c;
    }
    Poly operator * (Poly &b){
        Poly c;
        c.a.resize(a.size()+b.a.size()-1);
        for(int i=0;i<c.a.size();i++)c.a[i]=0;
        for(int i=0;i<a.size();i++)
            for(int j=0;j<b.a.size();j++)
                c.a[i+j]=c.a[i+j]+a[i]*b.a[j];
        while(c.a.size()>1 && isZero(*(c.a.end()-1)))c.a.erase(c.a.end()-1);
        return c;
    }
};
Poly Cn(ll k){
    Poly ans,t;
    ans=frac(1,1);
    for(ll i=0;i<k;i++){
        t=Poly((vector<type>){frac(-i,1),frac(1,1)});
        ans=ans*t;
        t=Poly((vector<type>){frac(1,i+1)});
        ans=ans*t;
    }
    return ans;
}
Poly sum(ll k){
    if(k==0)return Poly((vector<type>){frac(0,1),frac(1,1)});
    Poly ans=Cn(k+1),f=Cn(k),t,p;
    ans=ans+f;
    for(int i=1;i<k;i++){
        t=f(Poly((vector<type>){frac(i,1)}));
        ans=ans+t;
    }
    for(int i=0;i<k;i++){
        p=sum(i);
        t=f[i];
        t=t*p;
        ans=ans-t;
    }
    t=frac(1,1)/f[k];
    ans=ans*t;
    return ans;
}
Poly sum(Poly &f){
    Poly ans,t,p;
    ans=frac(0,1);
    for(int i=0;i<=f.n();i++){
        t=f[i];p=sum(i);t=t*p;
        ans=ans+t;
    }
    return ans;
}
Poly Lagrange(vector<type> x,vector<type> y){
    int n=x.size()-1;
    Poly ans;
```

```
    rep(k,0,n){
        Poly t,p;
        t=y[k];
        rep(j,0,n)if(j!=k){
            p=(vector<type>){frac(0,1)-x[j]/(x[k]-x[j]),frac(1,1)/(x[k]-x[j])};
            t=t*p;
        }
        ans=ans+t;
    }
    return ans;
}
int main(){
    cout<<sum(1);
    cout<<sum(2);
    cout<<sum(3);
    cout<<sum(4);
    return 0;
}
```

运行结果:

```
(1/2)x^2+(1/2)x^1+(0)
(1/3)x^3+(1/2)x^2+(1/6)x^1+(0)
(1/4)x^4+(1/2)x^3+(1/4)x^2+(0)x^1+(0)
(1/5)x^5+(1/2)x^4+(1/3)x^3+(0)x^2+(1/-30)x^1+(0)
```

1.2 一般多项式的前缀和

对于一般的多项式

$$f(x)=\sum_{i=0}^{k}a_ix^i$$

$$\sum_{i=1}^{n}f(i)=\sum_{i=0}^{k}\left(a_i\sum_{j=1}^{n}j^i\right)$$

多项式的前缀和依然是多项式，这是多么美妙的结论!

2.多项式插值算法

2.1 多项式插值的存在唯一性

多项式一直以来备受数学家们青睐，一方面它构造起来简单，另一方面它有非常美妙的性质，下面介绍多项式插值算法。

如果给定 n 个横纵坐标分别互不相同的点 $(x_i,y_i),i=1,2,\dots,n$，那么我们能否构造一个次数界为 n 的多项式函数，使得它的函数图像恰好经过这 n 个点? 答案是肯定的，而且这个多项式函数是唯一的，证明如下:

设存在这样的一个多项式

$$f(x)=\sum_{i=0}^{n-1}a_ix^i$$

根据构造条件，有

$$
\begin{cases}
f(x_1)=y_1\\
f(x_2)=y_2\\
\dots\\
f(x_n)=y_n
\end{cases}
$$

将上述线性方程组中的 $a_i, i=1,2,\dots,n$ ，视为未知量，其系数矩阵的行列式 &A& 恰好为范德蒙行列式，故

$$
det(A)=\prod_{1\le i<j\le n}(x_j-x_i)
$$

又 $x_i互不相同, i=1,2,\dots,n$，因此 $det(A)\neq 0$，方程组有唯一解

2.2 Lagrange多项式插值

那么问题就来了，如何求这个多项式呢？利用高斯消元解上述线性方程组是一个办法，算法的复杂度为 $O(n^3)$ ，其实这个多项式可以直接被构造出来

$$
f(x)=\sum_{i=1}^{n}\left(\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}\right)y_i
$$

这就是Lagrange插值公式，不难验证其次数至多为 $n-1$ ，且满足上述线性方程组，因此这就是我们要求的多项式。

参考代码：

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(ll i=a;i<=b;i++)
using namespace std;
typedef long long ll;
typedef long double type;
type lagrange(vector<type> x,vector<type> y,type X)
{
    int n=x.size()-1;
    type ans=0;
    rep(k,0,n)
    {
        type temp=y[k];
        rep(j,0,n)if(j!=k)temp*=(X-x[j])/(x[k]-x[j]);
        ans+=temp;
    }
    return ans;
}
int main()
{
    vector<type> x={0,1,2,3};
    vector<type> y={0,1,4,9};
    type X;
    while(cin>>X)cout<<lagrange(x,y,X)<<endl;
    return 0;
}
```

在ACM竞赛中，如果某个组合数学类题目刚好是输入一个整数 n ，输出一个多项式函数的值 $f(n)$ ，那么上述算法只需要放入某几项即可，不需要推导复杂的公式，例如2018年icpc南京现场赛的G题，将上述算法的除法修改为乘法逆元即可，至于最终的公式是啥以及如何推导，本文暂不讨论。

参考代码

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll mo=1e9+7;
ll fpow(ll a,ll b){
    ll ans=1;
    while(b>0){if(b&1)ans=ans*a%mo;b>>=1;a=a*a%mo;}
    return ans;
}
ll lagrange(vector<ll> x,vector<ll> y,ll X){
    auto p=y.begin();
    ll ans=0;
    for(auto k:x){
        ll a=*p++%mo,b=1;
        for(auto j:x)if(j!=k)a=(X-j)%mo*a%mo,b=(k-j)%mo*b%mo;
        ans=(ans+mo+a*fpow(b,mo-2)%mo)%mo;
    }
    return ans;
}
int main(){
    vector<ll> x={0,1,2,3,4};
    vector<ll> y={0,1,5,15,35};
    ll n;
    while(cin>>n)cout<<lagrange(x,y,n)<<endl;
    return 0;
}
```

其实在1.1的参考代码中已经给出了输出插值多项式的函数，可用如下方式调用

```cpp
int main(){
    vector<type> x={frac(0,1),frac(1,1),frac(2,1),frac(3,1),frac(4,1)};
    vector<type> y={frac(0,1),frac(1,1),frac(5,1),frac(15,1),frac(35,1)};
    Poly f=Lagrange(x,y);
    cout<<f;
    ll X;
    while(cin>>X)cout<<f(frac(X,1))<<endl;
    return 0;
}
```

运行结果

```
(1/24)x^4+(1/4)x^3+(11/24)x^2+(1/4)x^1+(0)
```

2.3 Newton多项式插值

Lagrange插值算法对于ACM竞赛中的相关题目来说可能已经足够了，但Lagrange插值算法最初并不是为了这么用的，它的主要用途是构造一个多项式来逼近另外一个函数，例如我们用计算机可能没办法计算三角函数 $sin(x)$ 的精确值，但是如果已知其中某些点的值，就可以构造这样的一个多项式来逼近 $sin(x)$ ，就可以计算其近似值，误差即为 $sin(x)$ 的泰勒展开式中的Lagrange余项。对于复杂的函数，如果增加一个插值点，那么多项式就需要重新构造，求解单点处的值的复杂度为 $O(n^2)$ ，于是数学家们想出了另一个算法---Newton多项式插值

首先定义差商：

零阶差商

$$F(x_i) = y_i$$

n 阶差商

$$F(x_1, x_2, \dots, x_{n+1}) = \frac{F(x_2, x_3, \dots, x_{n+1}) - F(x_1, x_2, \dots, x_n)}{x_n - x_1}$$

那么

$$f(x) = \sum_{i=1}^{n} \left(F(x_1, \dots, x_i) \prod_{j=1}^{i} (x - x_j) \right)$$

这样一来，这个算法就有了很好的继承性，每次添加一个插值点，复杂度为 $O(n)$ ，每次计算单点处的值，复杂度为 $O(n)$ （请参考秦九韶算法）。

参考代码（连续函数）

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(ll i=a;i<=b;i++)
using namespace std;
typedef long long ll;
typedef long double type;
class NewtonPoly{
    public:
    type f[105],d[105],x[105];
    ll n=0;
    void add(type X,type Y){
        x[n]=X,f[n]=Y;
        rep(i,1,n)f[n-i]=(f[n-i+1]-f[n-i])/(x[n]-x[n-i]);
        d[n++]=f[0];
    }
    type cal(type X){
        type ans=0,t=1;
        rep(i,0,n-1)ans+=d[i]*t,t*=X-x[i];
        return ans;
    }
}P;
int main(){
    P.add(0,0);
    P.add(1,1);
    P.add(2,5);
    P.add(3,15);
    P.add(4,35);
    type x;
    while(cin>>x)cout<<P.cal(x)<<endl;
    return 0;
}
```

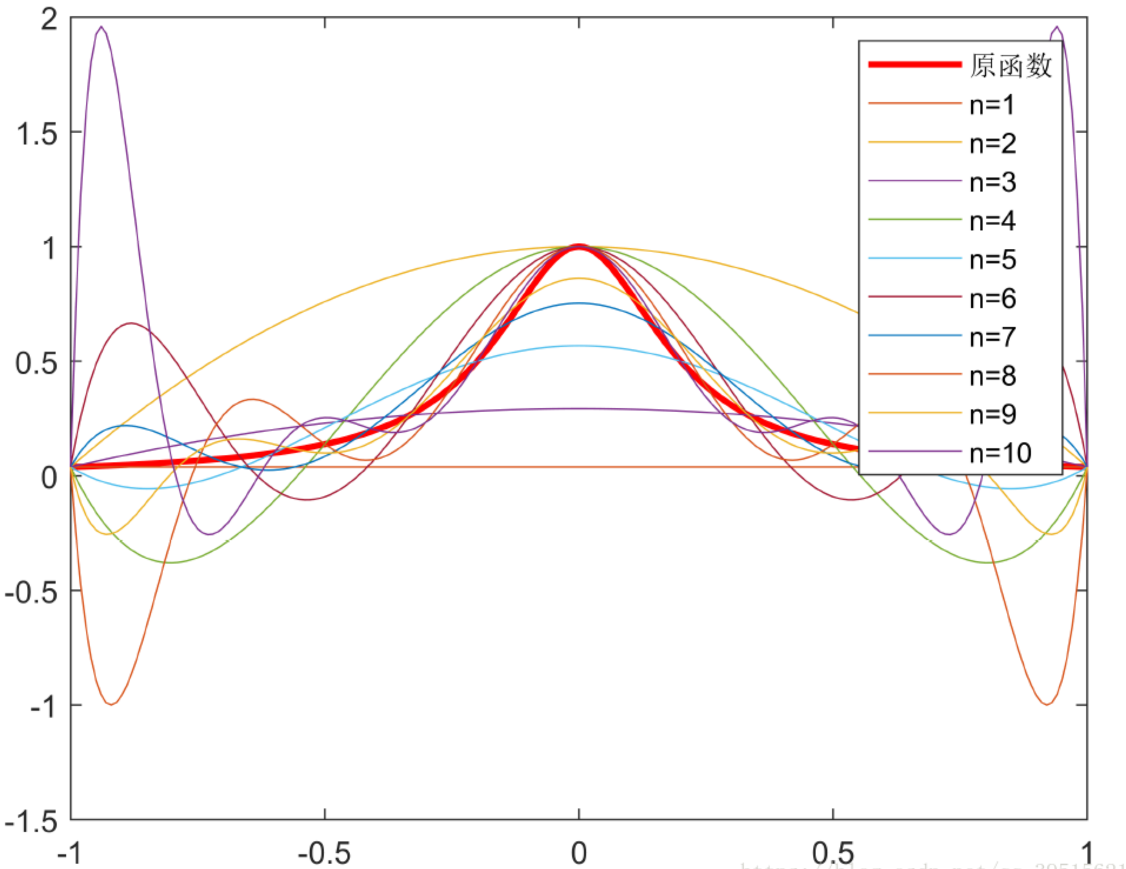
参考代码（离散函数，取余数，可用于ACM竞赛）

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(ll i=a;i<=b;i++)
using namespace std;
typedef long long ll;
const ll mo=1e9+7;
ll fpow(ll a,ll b){
    ll ans=1;
```

```
    while(b>0){if(b&1)ans=ans*a%mo;b>>=1;a=a*a%mo;}
    return ans;
}
class NewtonPoly{
    public:
    ll f[105],d[105],x[105],n=0;
    void add(ll X,ll Y){
        x[n]=X,f[n]=Y%mo;
        rep(i,1,n)f[n-i]=(f[n-i+1]-f[n-i])%mo*fpow((x[n]-x[n-i])%mo,mo-2)%mo;
        d[n++]=f[0];
    }
    ll cal(ll X){
        ll ans=0,t=1;
        rep(i,0,n-1)ans=(ans+d[i]*t)%mo,t=(X-x[i])%mo*t%mo;
        return ans+mo*(ans<0);
    }
}P;
int main(){
    P.add(0,0);
    P.add(1,1);
    P.add(2,5);
    P.add(3,15);
    P.add(4,35);
    ll x;
    while(cin>>x)cout<<P.cal(x)<<endl;
    return 0;
}
```

2.4 插值多项式的精度

还有一件事，插值点越多，精度就越高吗？事实可能出乎我们的预料，对于某些函数来讲，一昧地增加插值点得到个数，有时可能在边缘产生激烈的震荡，例如函数 $f(x)=\frac{1}{1+25x^2}$ ，这种激烈的震荡被称为龙格现象，解决这个问题的方法也很简单---分段，将区间切割成有限个小区间，每个小区间用三次多项式函数来逼近就足够了，这就是样条函数。事实上，如今设计师们常用的软件AI和PS中的钢笔工具，就是根据这个原理实现的。



2.5 高维插值整式

我们解决了一维的情况，二维的情况可由一维的Lagrange插值函数推广得来，更高维也类似

$$f(x,y)=\sum_{n=1}^{N}\sum_{m=1}^{M}\left(\prod_{i\neq n}\frac{x-x_i}{x_n-x_i}\right)\left(\prod_{j\neq m}\frac{y-y_j}{y_m-y_j}\right)z_{n,m}$$

参考代码（连续函数）

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(ll i=a;i<=b;i++)
using namespace std;
typedef long long ll;
typedef long double type;
type lagrange2(vector<type> x,vector<type> y,vector<vector<type> > z,type X,type
Y){
    int M=x.size()-1,N=y.size()-1;
    type ans=0;
    rep(m,0,M)rep(n,0,N){
        type t=z[m][n];
        rep(i,0,M)if(i!=m)t*=(X-x[i])/(x[m]-x[i]);
        rep(i,0,N)if(i!=n)t*=(Y-y[i])/(y[n]-y[i]);
        ans+=t;
    }
    return ans;
}
int main(){
    vector<type> x={1,2};
    vector<type> y={3,4};
    vector<vector<type> > z={{3,4},{6,8}};
    type X,Y;
```

```
    while(cin>>X>>Y)cout<<lagrange2(x,y,z,X,Y)<<endl;
    return 0;
}
```

参考代码（离散函数，取余数，可用于ACM竞赛）

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(ll i=a;i<=b;i++)
using namespace std;
typedef long long ll;
const ll mo=1e9+7;
ll fpow(ll a,ll b){
    ll ans=1;
    while(b>0){if(b&1)ans=ans*a%mo;b>>=1;a=a*a%mo;}
    return ans;
}
ll lagrange2(vector<ll> x,vector<ll> y,vector<vector<ll> > z,ll X,ll Y){
    ll M=x.size()-1,N=y.size()-1,ans=0;
    rep(m,0,M)rep(n,0,N){
        ll a=z[m][n]%mo,b=1;
        rep(i,0,M)if(i!=m)a=(X-x[i])%mo*a%mo,b=(x[m]-x[i])%mo*b%mo;
        rep(i,0,N)if(i!=n)a=(Y-y[i])%mo*a%mo,b=(y[n]-y[i])%mo*b%mo;
        ans=(ans+a*fpow(b,mo-2)%mo)%mo;
    }
    return ans+mo*(ans<0);
}
int main(){
    vector<ll> x={0,1,2,3,4,5,6,7,8,9};
    vector<ll> y={0,1,2,3,4,5,6,7,8,9};
    vector<vector<ll> > z={
        {0,0,0,0,0,0,0,0,0,0},
        {0,1,2,3,4,5,6,7,8,9},
        {-2,2,6,10,14,18,22,26,30,34},
        {-10,0,10,20,30,40,50,60,70,80},
        {-30,-10,10,30,50,70,90,110,130,150},
        {-70,-35,0,35,70,105,140,175,210,245},
        {-140,-84,-28,28,84,140,196,252,308,364},
        {-252,-168,-84,0,84,168,252,336,420,504},
        {-420,-300,-180,-60,60,180,300,420,540,660},
        {-660,-495,-330,-165,0,165,330,495,660,825}
    };
    ll X,Y;
    while(cin>>X>>Y){
        cout<<lagrange2(x,y,z,X,Y)<<endl;
    }
    return 0;
}
```