FFT——快速傅里叶变换

前言

一直想学FFT，当时由于数学基础太差，导致啥都学不懂。请教了机房里的几位学长大神，结果还是没太明白。因此下定决心写一篇关于“FFT”的文章，一篇起码我能看得懂的“FFT”。不过这好像并不是一件简单的事，因为首先我要学会“FFT”的原理。

建议同学们先自学一下“复数（虚数）”的性质、运算等知识，不然看这篇文章有很大概率看不懂。最后你就会发现，这不是一个“算法”的博客，而是一个数学的博客。

这是当时教我FFT的机房学长的博客：

Leo 的 《FFT与多项式乘法》

（温馨提示：整篇文章都不是很好理解，提醒同学们保持清醒的头脑。）

1.什么是FFT？

FFT，即为快速傅氏变换，是离散傅氏变换的快速算法，它是根据离散傅氏变换的奇、偶、虚、实等特性，对离散傅立叶变换的算法进行改进获得的。它对傅氏变换的理论并没有新的发现，但是对于在计算机系统或者说数字系统中应用离散傅立叶变换，可以说是进了一大步。——360百科

FFT（Fast Fourier Transformation）就是“快速傅里叶变换”的意思，它是一种用来计算DFT（离散傅里叶变换）和IDFT（离散傅里叶反变换）的一种快速算法。这种算法运用了一种高深的数学方式、把原来复杂度为的算法。

2.多项式乘法的朴素算法

假设我们现在不会FFT，我们是又是如何计算多项式乘法的呢？现在我们有两个关于x的二次多项式：





我命令K（x）为他们的乘积，则有：



如果在程序中，我们用一个数组来储存一个多项式的各个项的系数，我们如何去做这样一个复杂的乘法呢？

#include<iostream>

#include<vector>

#include<cstdlib>

using namespace std;

vector<double>ForceMul(vector<double>A,vector<double>B)//表示A,B两个多项式相乘的结果

{

vector<double>ans;

int aLen=A.size();//A的元素个数

int bLen=B.size();//B的元素个数

int ansLen=aLen+bLen-1;//ans的元素个数=A的元素个数+B的元素个数-1

for(int i=1;i<=ansLen;i++)//初始化ans

ans.push\_back(0);

for(int i=0;i<aLen;i++)

for(int j=0;j<bLen;j++)

ans[i+j]+=A[i]\*B[j];//A的i次项 与 B的j次项 相乘的结果 累加到ans的[i+j]次位

return ans;//返回ans

}

int main()

{

vector<double>A,B;

cout<<"input A:";

for(int i=0;i<3;i++)//从0次项开始输入A的各项系数

{

int x;

cin>>x;

A.push\_back(x);

}

cout<<"input B:";

for(int i=0;i<3;i++)//从0次项开始输入B的各项系数

{

int x;

cin>>x;

B.push\_back(x);

}

vector<double>C=ForceMul(A,B);//C=A与B暴力相乘

cout<<"output C:";

for(int i=0;i<5;i++)//从0次项开始输出C的各项系数

cout<<C[i]<<" ";

cout<<endl;

system("pause");

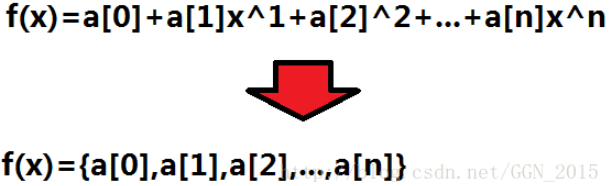
return 0;

}

这就是朴素算法,它的复杂度为O（lenA\*lenB）。如果lenA=lenB=10^5，程序时间就会爆掉，那么我们如何进行优化呢？

3.系数表示法与点值表示法

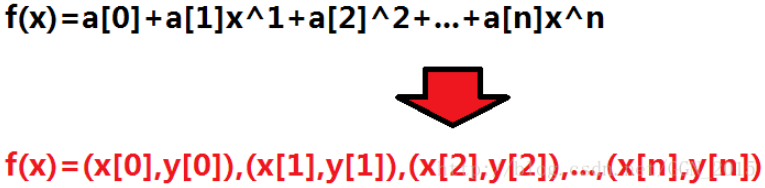
系数表表示法，就是用一个多项式的各个项的系数表示这个多项式，也就是我们平时所用的表示法。例如，我们可以这样表示：



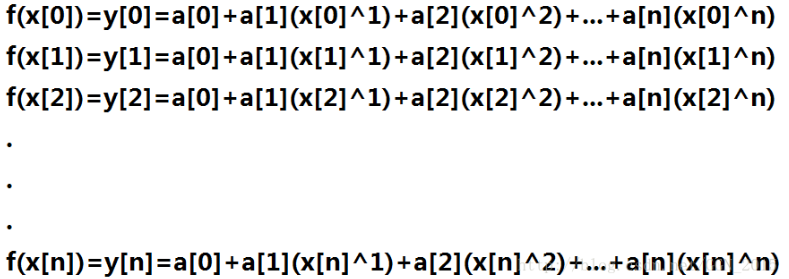
点值表示法，就是把这个多项式理解成一个函数，用这个函数上的若干个点的坐标来描述这个多项式。

（两点确定一条直线，三点确定一条抛物线…同理n+1个点确定一个n次函数，其原理来自于“高斯消元”，下文会有介绍。）

因此表示成这样：（注意：x[0]->x[n]是n+1个点）



为什么n+1个确定的点能确定一个唯一的多项式呢？你可以尝试着把这n+1个点的值分别代入多项式中：



你会得到n+1个方程，其中x[0~n]和y[0~n]是已知的，a[0~n]是未知的。n+1的未知数，n+1个方程所组成的方程组为“n+1元一次方程”，因为它是“一次方程”，所以（一般情况下，不考虑无解和无数解）可以通过“高斯消元”解得所有未知数唯一确定的值。也就是说，用点知表示法可以确定出 唯一确定的 系数表示法中的 每一位的 系数。

这种把一个多项式转化成“离散的点”表示的方法就叫做“DFT”（离散傅里叶变换）。

把“离散的点”还原回多项式的方法就叫做“IDFT”（离散傅里叶反变换）。

（请同学们重视上面的这两句话，因为这是我能想到的最好理解的解释方法了。）

有兴趣的可以看一下360百科给出的DFT定义：

离散傅里叶变换（Discrete Fourier Transform，缩写为DFT），是傅里叶变换在时域和频域上都呈离散的形式，将信号的时域采样变换为其DTFT的频域采样。在形式上，变换两端（时域和频域上）的序列是有限长的，而实际上这两组序列都应当被认为是离散周期信号的主值序列。即使对有限长的离散信号作DFT，也应当将其看作其周期延拓的变换。在实际应用中通常采用快速傅里叶变换计算DFT。——360百科

4.“复数”的引入

初中我们可能学过一些定理：比如”对于虚数的意义就相当于是“数字1”对于实数的意义。

这是百科对“虚数”的定义：

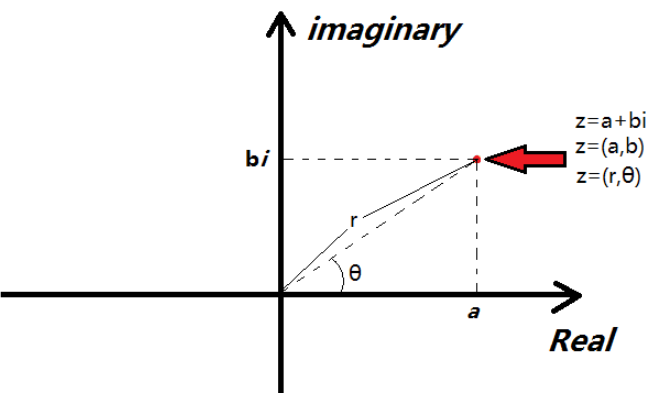
虚数可以指不实的数字或并非表明具体数量的数字。——360百科

在数学中，虚数就是形如a+b\*i的数，其中a,b是实数，且b≠0,i²= - 1。虚数这个名词是17世纪著名数学家笛卡尔创立，因为当时的观念认为这是真实不存在的数字。后来发现虚数a+b\*i的实部a可对应平面上的横轴虚部b与对应平面上的纵轴，这样虚数a+b\*i可与平面内的点(a,b)对应。——360百科

然后是百科对“复数”的定义：

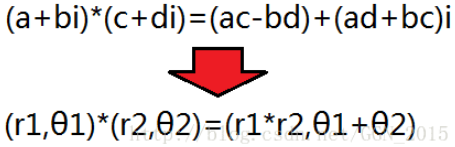
复数x被定义为二元有序实数对(a,b) ，记为z=a+bi,这里a和b是实数，i是虚数单位。在复数a+bi中，a=Re(z)称为实部，b=Im(z)称为虚部。当虚部等于零时，这个复数可以视为实数;当z的虚部不等于零时，实部等于零时，常称z为纯虚数。复数域是实数域的代数闭包，也即任何复系数多项式在复数域中总有根。 复数是由意大利米兰学者卡当在十六世纪首次引入，经过达朗贝尔、棣莫弗、欧拉、高斯等人的工作，此概念逐渐为数学家所接受。

正如上文所说，一个复数可以看成是“复平面”上的一个点。复平面就是以实数部为x轴，以虚部为y轴所组成的类似“直角坐标系”的一个平面坐标体系。同样，我们也可以用“极坐标”来表示一个平面中的点。



上图就是在这个复平面上的一个点的三种表示方法。

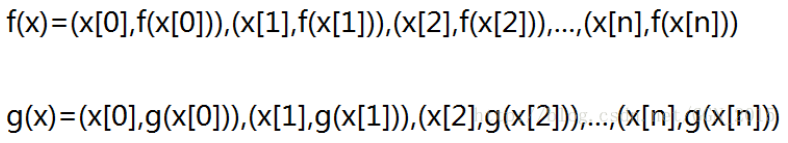
然后，思考一个简单的问题：两个复数的乘法有没有某种特定的几何意义？（只是一个数学性质，在此不进一步深究，可用三角函数证明。）



两复数相乘，“长度相乘，极角相加”。不难想象如果两个复数它们到“坐标原点”的距离都是1，那么它们的乘积到坐标原点的距离还是一，只不过是绕着原点进行了旋转。

5.单位复根

现在，回到我们刚才讲到的“点值表示法”的问题，考虑这样一个问题，如果我有两个用点值表示的多项式，如何表示它们两个多项式的乘积呢？（假设这两个多项式选取的所有点的x值恰好相同。）

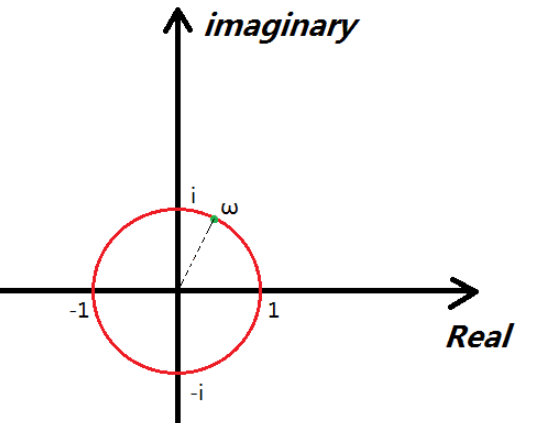


如果的操作。）



但是我们要的是系数表达式，而不是点制表达式，如果用高斯消元暴力地去解一个“n+1元方程组”恐怕就要把时间复杂度拉回时会浪费大量的时间。这个数学运算看似是没有办法加速的，而实际上我们可以找到一种神奇的“x值”，带进去之后不用反复地去做无用n次方操作。我能想到的第一个数就是“1”，因为1的几次方都是1。然后我能想到的数就是“-1”，因为“-1”的平方是1。把这样的数带进去就可以减少我们的运算次数。

但是问题又来了，我们只有“-1”和“1”两个数，但是我们要至少带进去n+1个不同的数才能进行系数表示。考虑一个问题：也许“虚数”可能会帮上我们大忙。我们需要的是满足“”的数（k为整数），然后我们就会发现“i”好像也满足这个条件：i\*i=-1，（i\*i）\*（i\*i）=1=i^4，当然“-i”也有这个性质。然而仅仅4个数还是不能满足我们的需求。

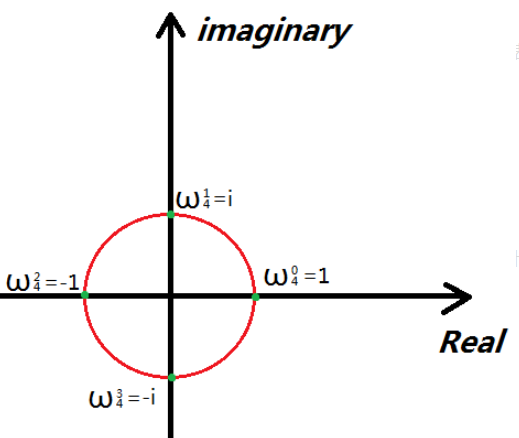


看上图中的红圈，红圈上的每一个点距原点的距离都是1个单位长度，所以说如果说对这些点做k次方运算，它们始终不会脱离这个红圈。因为它们在相乘的时候r始终=1，只是θ的大小在发生改变。而这些点中有无数个点经过k次方之后可以回到“1”。因此，我们可以把这样的一组神奇的x带入函数求值。

像这种能够通过k次方运算回到“1”的数，我们叫它“复根”用“ω”表示。如果”:



其中“n”就是一个序号数，我们把所有的“负根”按照极角大小逆时针排序从零开始编号。以“四次负根” ””为例：



你会发现：其实k次负根就相当于是给图中的圆周平均分成k个弧，弧与弧之间的端点就是“复根”，另外”为“单位复根”。

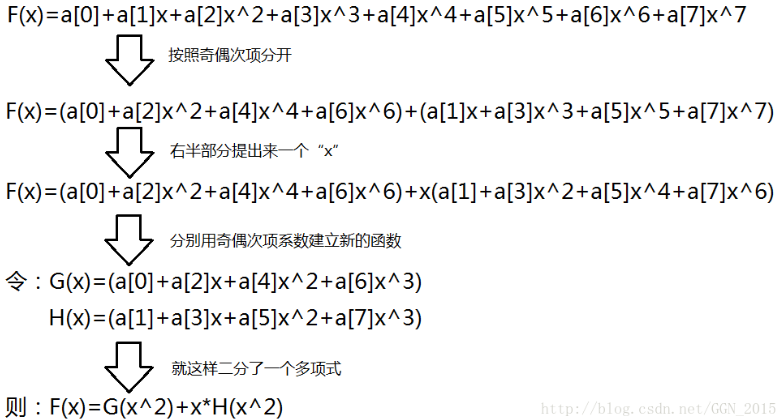
其实，我们用“表示的是“单位复根”的“1次方”也就是它本身,其他的就叫做k次单位复根的n次方。



k次单位复根

6.FFT的主要流程之DFT

说了这么多了，终于说到FFT了。FFT运用到了一种分治的思想，分治地去求当x=ω(k)[n]时整个多项式的值（永远都不要忘了每一个步骤的目的是什么）。你可以把一个多项式分成奇数次数项，和偶数次数项两部分，然后再用分治的思想去处理它的“奇数次项”和“偶数次项”。



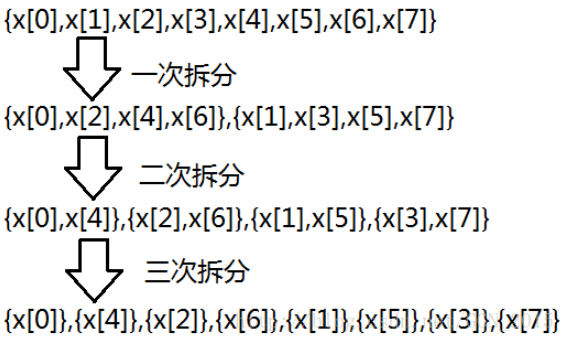
我们用DFT(F(x))[k]表示当x=ω^k时F(x)的值，所以有：DFT(F(x))[k]=DFT(G(x^2))[k]+ω^k\*DFT(H(x^2))，也就是：



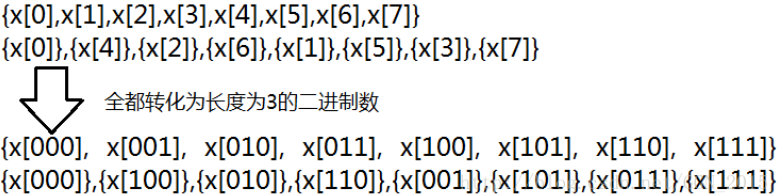
（把当前单位复根的平方分别以DFT的方式带入G函数和H函数求值。）

但是这个二分最大的局限就是只能处理长度为2的整数次幂的多项式，因为如果长度不为的整数次幂，二分到最后就会出现左半部分右半部分长度不一致的情况（导致程序取不到系数而爆炸），所以我们在做第一次DFT之前一定要把这个多项式补成长度为2^n（n为整数）的多项式（补上去的高次项系数为“0”），长度为2^n的多项式的最高次项为2^n-1次项。当我们向这个式子中“带入数值”的时候，一定要保证我带入的每个数都是不一样的，所以要带入1的2^n的单位复根的各个幂（因为1的2^n复根恰好有2^n个）。

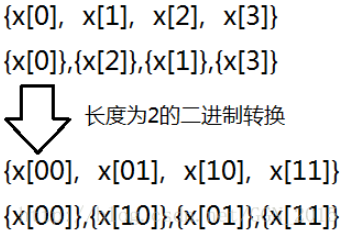
但是这个算法还需要从“分治”的角度继续优化。我们每一次都会把整个多项式的奇数次项和偶数次项系数分开，一只分到只剩下一个系数。但是，这个递归的过程需要更多的内存。因此，我们可以先“模仿递归”把这些系数在原数组中“拆分”，然后再“倍增”地去合并这些算出来的值。然而我们又要如何去拆分这些数呢？



貌似并没有什么规律可循，但实际上你可要看仔细了，把这些下标都转化成二进制看看吧：



是不是发现了一些神奇的特点：“拆分”之后的序列的下标恰好为长度为3位的二进制数的翻转。也就是说我们对原来的每个数的下标进行长度为三的二进制翻转就是新的下标。而为什么是长度为3呢？因为8=2^3。为了证明这一点，我们可以再举一个简单例子（它还是成立的）：



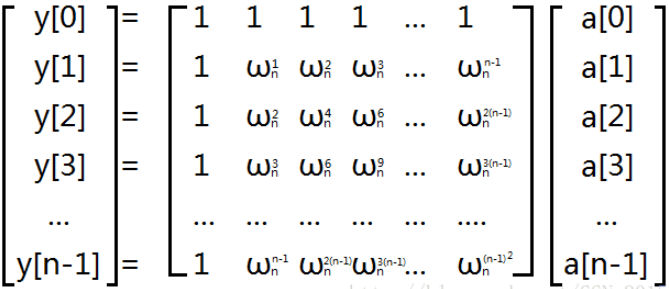
（一个数学性质，在此不再证明。我感觉这个原理有点像是“基数排序”，感兴趣的同学可以去看看。）

关于二进制翻转是如何实现的，本文中并没有介绍，强烈建议看一下这篇文章：

学习链接：补充——FFT中的二进制翻转问题

7.FFT的主要流程之IDFT

我们先整理一下思路，IDFT是做什么的？IDFT（傅里叶反变换）就是把一个用“点值表示法”表示的多项式，转化成一个用“系数表示法”表示的多项式，但是这似乎并不是很容易。然而其实我们刚刚恰好做了一些非常机智的事情——把“单位复根”的若干次方带入了原多项式。我们可以表示一下这些多项式（这里使用一个矩阵表示，不会的建议自学）。



如果我们想把这个表达式还原成只含有“a系数”的矩阵，那么就要在中间那个“巨大的矩阵”身上乘上一个它的“反矩阵”（反对称矩阵）就可以了。这个矩阵的中有一种非常特殊的性质，对该矩阵的每一项取倒数，再除以n就可以得到该矩阵的反矩阵。而如何改变我们的操作才能使计算的结果文原来的倒数呢？这就要看我们求“单位复根的过程了”：根据“欧拉函数”e^iπ=-1，我么可以得到e^2πi=1。如果我要找到一个数，它的k次方=1，那么这个数ω[k]=e^(2πi/k)（因为(e^(2πi/k))^k=e^(2πi)=1）。而如果我要使这个数值变成“1/ω[k]”也就是“(ω[k])^-1”，我们可以尝试着把π取成-3.14159…，这样我们的计算结果就会变成原来的相反数，而其它的操作过程与DFT是完全相同的（这真是极好的）。我们可以定义一个函数，向里面掺一个参数“1”或者是“-1”，然后把它乘到“π”的身上。传入“1”就是DFT，传入“-1”就是IDFT，十分的智能。

机房学长的代码写得实在是太好了Orz，忍不住引用了一下：

下面的代码来自 Leo的 《FFT与多项式乘法》

欢迎同学们一同Orz ↑

这是整个代码的一个局部（出于礼貌，保留了原有的代码格式，但是加了一些注释）：

int rev[maxl];

void get\_rev(int bit)//bit表示二进制位数,计算一个数在二进制翻转之后形成的新数

{

for(int i=0;i<(1<<bit);i++)

rev[i]=(rev[i>>1]>>1)|((i&1)<<(bit-1));

}

void fft(cd \*a,int n,int dft)//n表示我的多项式位数

{

for(int i=0;i<n;i++) if(i<rev[i]) swap(a[i],a[rev[i]]);

//中间的那个if保证了每个数做多只被交换了1次

//如果不写那么会有一些数被交换两次,导致最终的位置没有变

for(int step=1;step<n;step<<=1)//模拟一个合并的过程

{

cd wn=exp(cd(0,dft\*PI/step));//计算当前单位复根

for(int j=0;j<n;j+=step<<1)

{

cd wnk(1,0);//计算当前复根

for(int k=j;k<j+step;k++)

{//蝴蝶操作

cd x=a[k];

cd y=wnk\*a[k+step];

a[k]=x+y;//这就是上文中F(x)=G(x)+ωH(x)的体现

a[k+step]=x-y;

//后半个“step”中的ω一定和“前半个”中的成相反数

//“红圈”上的点转一整圈“转回来”，转半圈正好转成相反数

wnk\*=wn;

}

}

}

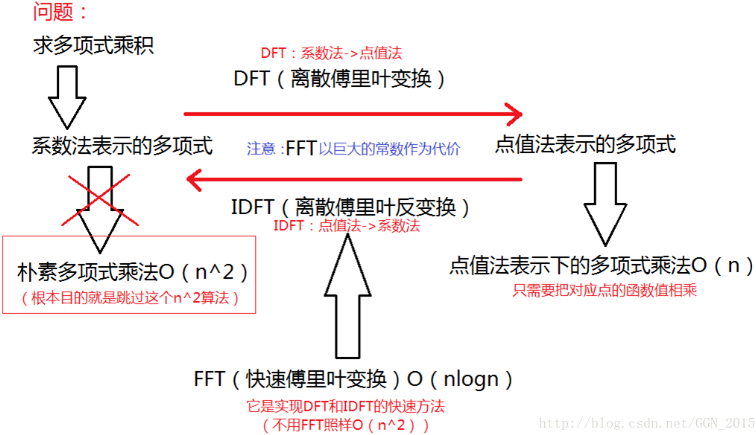
if(dft==-1) for(int i=0;i<n;i++) a[i]/=n;

//考虑到如果是IDFT操作,整个矩阵中的内容还要乘上1/n

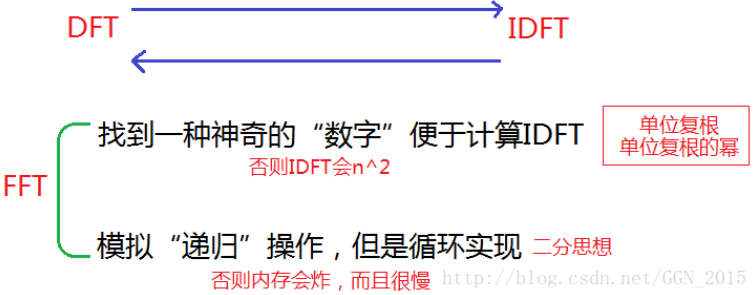
}

8.后记

最后总结一下FFT的优化思想：



然后是FFT的优化理念：



（完全自创，如有雷同，纯属巧合，尽管并没什么用。）

FFT其实还可以用来计算BIGNUM乘法，因为我们可以把一个长整数理解成a[0]+a[1]\*10+a[2]\*10^2+…+a[n]\*10^n。把“10”当成未知数，这个多项式每一个次方项的系数就是BIGNUM每一数位上的数。而这时，数组长度“n”就不能单纯的取这个十进制数的长度，而要取大于等于两个十进制数长度加和的最小的2的正整数次幂。因为我们要保证DFT得到的离散点的个数足够表示我最终生成的新多项式（也就是取的点的个数要大于等于这个结果多项式的长度）。

总之，这真的是一个很好的算法，但是要注意，除题中的数据范围（多项式长度）超过了10^4，否则暴力是可以解决的。而且FFT常数巨大，请同学们一定要慎用！慎用！慎用！

写了一整天，可算是把这篇长篇大论的博客写完了。FFT就是一个典型的用数学方法对问题实现优化的方法。DFT对数学基础要求很高，但是信息学与数学的联系是相当紧密的，所以这种情况时常会出现。数学不好的同志们一定要加油哦~

赶稿匆忙，如有谬误，望同学们谅解。

2017.11.19【补】第一次自己手写FFT，感觉很开心

#include<cstdio>

#include<cstdlib>

#include<cmath>

#include<algorithm>

#include<cstring>

#include<complex>

using namespace std;

typedef complex<double> cd;//复数类的定义

const int maxl=2094153;//nlogn的最大长度(来自leo学长的博客)

const double PI=3.14159265358979;//圆周率,不解释

cd a[maxl],b[maxl];//用于储存变换的中间结果

int rev[maxl];//用于储存二进制反转的结果

void getrev(int bit){

for(int i=0;i<(1<<bit);i++){//高位决定二进制数的大小

rev[i]=(rev[i>>1]>>1)|((i&1)<<(bit-1));

}//能保证(x>>1)<x,满足递推性质

}

void fft(cd\* a,int n,int dft){//变换主要过程

for(int i=0;i<n;i++){//按照二进制反转

if(i<rev[i])//保证只把前面的数和后面的数交换,(否则数组会被翻回来)

swap(a[i],a[rev[i]]);

}

for(int step=1;step<n;step<<=1){//枚举步长的一半

cd wn=exp(cd(0,dft\*PI/step));//计算单位复根

for(int j=0;j<n;j+=step<<1){//对于每一块

cd wnk(1,0);//!!每一块都是一个独立序列,都是以零次方位为起始的

for(int k=j;k<j+step;k++){//蝴蝶操作处理这一块

cd x=a[k];

cd y=wnk\*a[k+step];

a[k]=x+y;

a[k+step]=x-y;

wnk\*=wn;//计算下一次的复根

}

}

}

if(dft==-1){//如果是反变换,则要将序列除以n

for(int i=0;i<n;i++)

a[i]/=n;

}

}

int output[maxl];

char s1[maxl],s2[maxl];

int main(){

scanf("%s%s",s1,s2);//读入两个数

int l1=strlen(s1),l2=strlen(s2);//就算"次数界"

int bit=1,s=2;//s表示分割之前整块的长度

for(bit=1;(1<<bit)<l1+l2-1;bit++){

s<<=1;//找到第一个二的整数次幂使得其可以容纳这两个数的乘积

}

for(int i=0;i<l1;i++){//第一个数装入a

a[i]=(double)(s1[l1-i-1]-'0');

}

for(int i=0;i<l2;i++){//第二个数装入b

b[i]=(double)(s2[l2-i-1]-'0');

}

getrev(bit);fft(a,s,1);fft(b,s,1);//dft

for(int i=0;i<s;i++)a[i]\*=b[i];//对应相乘

fft(a,s,-1);//idft

for(int i=0;i<s;i++){//还原成十进制数

output[i]+=(int)(a[i].real()+0.5);//注意精度误差

output[i+1]+=output[i]/10;

output[i]%=10;

}

int i;

for(i=l1+l2;!output[i]&&i>=0;i--);//去掉前导零

if(i==-1)printf("0");//特判长度为0的情况

for(;i>=0;i--){//输出这个十进制数

printf("%d",output[i]);

}

putchar('\n');

return 0;

}

[2017.12.1]在此补充一下NTT（快速数论变换）的多项式乘法的代码。

#include<cstdio>

#include<cstdlib>

#include<algorithm>

#include<cmath>

//#include<complex>

using namespace std;

//typedef complex<double> cd;

typedef long long LL;

void exgcd(int a,int b,int& x,int& y){

if(b==0){

x=1;

y=0;

return;

}

int x0,y0;

exgcd(b,a%b,x0,y0);

x=y0;y=x0-int(a/b)\*y0;

}

int Inv(int a,int p){

int x,y;

exgcd(a,p,x,y);

x%=p;

while(x<0)x+=p;

return x;

}

int qpow(int a,int b,int p){

if(b<0){

b=-b;

a=Inv(a,p);

}

LL ans=1,mul=a%p;

while(b){

if(b&1)ans=ans\*mul%p;

mul=mul\*mul%p;

b>>=1;

}

return ans;

}

#define maxn (65537\*2)

const int MOD=479\*(1<<21)+1,G=3;

int rev[maxn];

void get\_rev(int bit){

for(int i=0;i<(1<<bit);i++){

rev[i]=(rev[i>>1]>>1)|((i&1)<<(bit-1));

}

}

//from internet

//for(int i=0; i<NUM; i++)

//{

// int t = 1 << i;

// wn[i] = quick\_mod(G, (P - 1) / t, P);

//}

LL a[maxn],b[maxn];

void ntt(LL\* a,int n,int dft){

for(int i=0;i<n;i++){

if(i<rev[i])

swap(a[i],a[rev[i]]);

}

for(int step=1;step<n;step<<=1){

LL wn;

wn=qpow(G,dft\*(MOD-1)/(step\*2),MOD);

for(int j=0;j<n;j+=step<<1){

LL wnk=1;//这里一定要用long long不然会迷之溢出

for(int k=j;k<j+step;k++){

LL x=a[k]%MOD,y=(wnk\*a[k+step])%MOD;//这里也要用long long

a[k]=(x+y)%MOD;

a[k+step]=((x-y)%MOD+MOD)%MOD;

wnk=(wnk\*wn)%MOD;

}

}

}

if(dft==-1){

int nI=Inv(n,MOD);

for(int i=0;i<n;i++)

a[i]=a[i]\*nI%MOD;

}

}

#include<cstring>

char s1[maxn],s2[maxn];

int main(){

//scanf("%\*d");

scanf("%s%s",s1,s2);

int l1=strlen(s1),l2=strlen(s2);

for(int i=0;i<l1;i++)a[i]=s1[l1-i-1]-'0';

for(int i=0;i<l2;i++)b[i]=s2[l2-i-1]-'0';

int bit,s=2;

for(bit=1;(1<<bit)<(l1+l2-1);bit++)s<<=1;

get\_rev(bit);ntt(a,s,1);ntt(b,s,1);

for(int i=0;i<s;i++)

a[i]=a[i]\*b[i]%MOD;

ntt(a,s,-1);

for(int i=0;i<s;i++){

a[i+1]+=a[i]/10;

a[i]%=10;

}

int cnt=s;

while(cnt>=0 && a[cnt]==0)cnt--;

if(cnt==-1)printf("0");

for(int i=cnt;i>=0;i--){

printf("%d",a[i]);

}

putchar('\n');

return 0;

}

<https://blog.csdn.net/dt_kang/article/details/79687208>