# 快速傅里叶变换：

## 离散傅里叶变换：

序列xn=(x0 x1 x2….xn-1)要求得到坐标变换后的序列F

求得另一个序列F=T\*x

F和x都是一维向量，T是个n\*n的矩阵，求法如下：

T=

W==cos(2π/n)-i\*sin(2π/n) //是傅里系数的复数形式，由复变函数里的欧拉公式formula推导得来

求出T后与x做矩阵乘法，得到F，这就是DFT(离散傅里叶变换)，很容易实现

若已知F求x，用F\*，注意到T是个范德蒙矩阵，求逆有规律，结果是

= \*1/n

W==cos(2π/n)+i\*sin(2π/n)

已知F 求x, x= \* F，这是离散傅里叶变换的逆变换

求一维数组和n\*n矩阵T相乘复杂度n\*n;算矩阵T又是n\*n，总时间复杂度O()

## 快速傅里叶变换FFT：

快速傅里叶变换是离散傅里叶变换的快速算法，

快速傅里叶变换是把矩阵分解成2个一半大小，n=n/2

结果是n/4+n/4=n/2,仅相当于原来的一半，因此如此递归下去，复杂度降到了n\*logn

设N是点的个数，且fft里N是2的次方数(不够补0)

以下k代表第几个点，也就是最后F数组的下标

F[k]=

按照x下标奇数偶数分类有：

=

+

提出一个

=+

利用那个W的性质

回来看那个W是怎么求得,设==cos(2π/n)-i\*sin(2π/n)

=== cos(2π/(n/2))-i\*sin(2π/(n/2))

=

推广=

所以接着上边

=+

=+

可以看出正好是所有偶数共N/2个点的离散傅里叶变换，设它是E[k]；正好是所有奇数共N/2个点的离散傅里叶变换，设它是O[k]。

则有：F[k]= E[k] +O[k] , 0<=k<N

到这里又用W性质，它具有周期性

==, 证明参考求W的公式,以及复变函数基本公式。

所以得到E[k]=E[k+N/2],O[k]=O[k+N/2](这里周期是N/2)

那么F[k+N/2]= E[k+N/2] +O[k+N/2] =

E[k] +O[k]

利用周期性，只需要计算一下，就求出后面一半的FFT, 时间变成了一半。

如此做，把E[K]和O[k]也就是偶数点，奇数点的DFT再各自当成F数组，去做FFT，如此递归下去，直到分成1个点为止，任何1个单独的点的DFT就等于它本身。

由于是FFT是对2的整数次方操作，必定可以保证可以分到各自单独的点

## FFT的常数优化：

对于代码来说FFT写的好坏常数相差很大，我也是看别人代码才优化到很快的，有一个很好的验证方式是51nod大数乘法那道题，很快的代码应该小于100ms过

* 倒序优化：简单来说就是把奇数位放一起，偶数位放一起。

对数组求倒序使用雷德算法，可以用如下代码表示，for(i=0,j=0;i<N;++i){

if(i<j)swap(x[i],x[j]);

for(k=N/2;(j^=k)<k;k>>=1);

} //雷德算法在下一节介绍

实际上这是一个双重for循环复杂度是NlogN的，但对于多次求同一个长度为N的数组的倒序，还有优化空间，这里因为x[]是一个等待FFT的数组，每次x[]肯定是不同的，没法预处理x数组，但是每次枚举了x[i]后让x[i]和x[j]交换，对于相同的N，可以用预处理的方式得到每个i对应j的值。

## 雷德算法

使用雷德算法实现倒位序：

  对于自然顺序（二进制）我们是在低位加 1 得到下一位数，对于倒位序我们是在高位加 1 向低位进位。比如已知一个倒位序数是J求其下一个倒位序数，N位总数 ，把J与N/2比较若J<N/2则J的最高位为 0 ，把最高位置 1 ，就得到了J的下一个倒位序数；若J>=N/2则说明J的最高位为1 ，把最高位置0 ，比较次高位，若次高位为0 ，则把次高位置1，得到J的下一个倒位序，若次高位为1  ， 则把次高位置0,以此类推.

以N = 8 为例：

倒位数顺序                  倒位数                  十进制

    000                           000                          0

    001                           100                          4

    010                            010                         2

    011                            110                         6

    100                            001                         1

    101                            101                         5

    110                            011                         3

111                            111                         7

达到的效果： 把0到N-1得2进制左右对称的换一下,使得偶数在前半部分，奇数在后半部分，算法时间复杂度为O(n\*logn)

## FFT解决问题

**卷积定义:**

已知x(t),h(t)是t的函数，若y(t)满足下面的等式，则称y是函数x和h的卷积。

连续：formula

离散：

计算机中一般用离散的公式，且i是有限的，如果从1到t,暴力计算仍要O(n^2)复杂度。

如果设fft(x)代表对x进行fft，ifft(x) 代表对x进行逆fft后，可以证明：//证明略

把fy[]序列做一次逆fft就是y[]

如果求fft和ifft时用快速傅里叶变换算法，则总复杂度为O(nlogn)，比暴力计算的O(n\*n)快

# 验证：

举个例子（下面一对数代表一个复数a+bi）

FFT是浮点数运算会有精度损失，结果只要近似相等就好，误差在

1e-5以内可以接受，计算比较麻烦，借助DFT程序来验证

N=8

i=0 1 2

i=1 3 4

i=2 5 6

i=3 7 8

i=4 9 -1

i=5 10 -2

i=6 16 -3

i=7 20 -4

## 1.运行dft程序求原数据

8

1 2

3 4

5 6

7 8

9 -1

10 -2

16 -3

20 -4

得到：

71.0000000000 10.0000000000

17.9705603301 23.8994961636

-13.0000029474 11.9999980708

-8.5147211277 10.3847734876

-9.0000000000 -2.0000038585

-15.9705627143 4.1004981248

-8.9999914792 -16.0000054662

-25.4852601561 -26.3847821026

## 2.运行N=8的分解，偶数i=0 2 4 6 的dft:

4

1 2

5 6

9 -1

16 -3

得到

31.0000000000 4.0000000000

0.9999987942 13.9999999196

-11.0000000536 -2.0000018756

-16.9999967042 -8.0000026527

## 3. 运行N=8的分解，奇数I= 1 3 5 7 的dft：

4

3 4

7 8

10 -2

20 -4

得到：

40.0000000000 6.0000000000

4.9999984727 19.0000000000

-14.0000000000 -2.0000025187

-18.9999960611 -7.0000032154

## 4.验证当前正确性：

W(N,K)代表

F[k]= E[k] +O[k];

F[0]= (31.0000000000 4.0000000000)+W(8,0)\*( 40.0000000000 6.0000000000)

=(71.0000000000 10.0000000000)

F[1]= (0.9999987942 13.9999999196)+W(8,1)\*( 4.9999984727 19.0000000000)

=(17.9705603301 23.8994961636)

F[2]= (-11.0000000536 -2.0000018756)+W(8,2)\*( -14.0000000000 -2.0000025187)

=(-13.0000029474 11.9999980708)

F[3]= (-16.9999967042 -8.0000026527)+W(8,3)\*( -18.9999960611 -7.0000032154)

=(-8.5147211277 10.3847734876)

K>=N/2时候，E和O里面的k=k-N/2,原因参见周期规律

F[4]=E(4)+W(8,4)\*O(4)= E(0)+W(8,4)\*O(0)

F[4]= (31.0000000000 4.0000000000)+W(8,4)\*( 40.0000000000 6.0000000000)

=(-8.9999996785 -2.0000021436)

F[5]=E(5)+W(8,5)\*O(5)= E(1)+W(8,5)\*O(1)

F[5]= (0.9999987942 13.9999999196)+W(8,5)\*( 4.9999984727 19.0000000000)

=(-15.9705622112 4.1005027662)

F[6]= (-11.0000000536 -2.0000018756)+W(8,6)\*( -14.0000000000 -2.0000025187)

=(-8.9999964095 -16.0000017148)

F[7]= (-16.9999967042 -8.0000026527)+W(8,7)\*( -18.9999960611 -7.0000032154)

=(-25.4852712954 -26.3847792477)

## 5.进一步分解：把N=4在分解

## N==4时候的也不用DFT求了

* 运行N=4\_E的 偶数i=0 ,4分解

2

1 2

9 -1

得到

10.0000000000 1.0000000000

-8.0000000536 2.9999995177

* 运行N=4\_E 奇数i=2,6分解

2

5 6

16 -3

得到

21.0000000000 3.0000000000

-11.0000001608 8.9999991426

* 验证N=4\_E的DFT是否等于：

现在的F0代表N=4\_E的DFT

F0[0]=( 10.0000000000 1.0000000000)+W(4,0) (21.0000000000 3.0000000000)

=(31.0000000000 4.0000000000)

F0[1]=( -8.0000000536 2.9999995177)+W(4,1) (-11.0000001608 8.9999991426)

=(0.9999987943 13.9999999197)

F0[2]=( 10.0000000000 1.0000000000)+W(4,2) (21.0000000000 3.0000000000)

=(-10.9999998392 -2.0000011254)

F0[3]=( -8.0000000536 2.9999995177)+W(4,3) (-11.0000001608 8.9999991426)

=(-16.9999983120 -8.0000013666)

* 运行N=4\_O的 偶数i=1 ,5分解

2

3 4

10 -2

得到

13.0000000000 2.0000000000

-7.0000001072 5.9999994641

* 运行N=4\_O 奇数i=3,7分解

2

7 8

20 -4

得到

27.0000000000 4.0000000000

-13.0000002144 11.9999989282

* 验证N=4\_O 的DFT是否等于：

现在的F1代表N=4\_O的DFT

F1[0]=( 13.0000000000 2.0000000000)+W(4,0) (27.0000000000 4.0000000000)

=(40.0000000000 6.0000000000)

F1[1]=( -7.0000001072 5.9999994641)+W(4,1) (-13.0000002144 11.9999989282)

=(4.9999984727 19.0000000000)

F1[2]=( 13.0000000000 2.0000000000)+W(4,2) (27.0000000000 4.0000000000)

=(-13.9999997856 -2.0000014469)

F1[3]=( -7.0000001072 5.9999994641)+W(4,3) (-13.0000002144 11.9999989282)

=(-18.9999979904 -7.0000017149)

## 6.上面的步骤都是对的，最后当然是，把等于N=2的情况分解，而任何N==1也就是单独点的DFT是它自己，在求解N=2每组数的DFT时候，F[k]=E[k]+W(2,k)\*O(k)=A[i]+ W(2,k)\*A[i],A[i]之中的i怎么求，我们计算的时候当然是希望把可以FFT的数对两两排好，这个算法是求倒位序

参见另一个word<雷德算法.docx>

举个例子还是：

最后一组来说

2

7 8

20 -4

应该是

27.0000000000 4.0000000000

-13.0000002144 11.9999989282

把2个数 分奇数偶数项

偶数项 DFT 是 7 8

奇数项DFT是 20 -4

F[0]=(7,8)+W(2,0)\*(20,-4)= 27.0000000000 4.0000000000

F[1]= (7,8)+W(2,1)\*(20,-4)= -13.0000002144 11.9999989282

确实正确；

到这里FFT的算法步骤就演示出来了。把它变得不再抽象

就差代码实现，至于逆变换，道理差不多，把W的正负符号改一下，最后F数组除以N就可以了