# **[快速傅里叶变换(FFT)详解](https://www.cnblogs.com/zwfymqz/p/8244902.html)**

**目录**

* [前言](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label0)
* [多项式](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label1)
  + [系数表示法](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label1_0)
  + [点值表示法](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label1_1)
* [复数](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2)
  + [向量](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2_0)
  + [圆的弧度制](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2_1)
  + [平行四边形定则](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2_2)
  + [复数](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2_3)
  + [运算法则](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2_4)
  + [单位根](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2_5)
  + [单位根的性质](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label2_6)
* [快速傅里叶变换](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label3)
* [快速傅里叶逆变换](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label4)
* [理论总结](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label5)
* [递归实现](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label6)
* [迭代实现](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_label7)

本文只讨论FFT在信息学奥赛中的应用

文中内容均为个人理解，如有错误请指出，不胜感激

[回到顶部](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_labelTop)

## 前言

先解释几个比较容易混淆的缩写吧

DFT：离散傅里叶变换—>O(n2)计算多项式乘法

FFT：快速傅里叶变换—>O(n∗log⁡(n)计算多项式乘法

FNTT/NTT：快速傅里叶变换的优化版—>优化常数及误差

FWT：快速沃尔什变换—>利用类似FFT的东西解决一类卷积问题

MTT：毛爷爷的FFT—>非常nb/任意模数

FMT 快速莫比乌斯变化—>感谢stump提供

[回到顶部](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_labelTop)

## 多项式

### **系数表示法**

设A(x)表示一个n−1次多项式

则

例如：A(3)=2+3\*x+x2

利用这种方法计算多项式乘法复杂度为O(n2)

（第一个多项式中每个系数都需要与第二个多项式的每个系数相乘）

### **点值表示法**

将n互不相同的x带入多项式，会得到n个不同的取值y

则该多项式被这n个点(x1,y1),(x2,y2),…,(xn,yn)唯一确定

其中

例如：上面的例子用点值表示法可以为(0,2),(1,6),(2,12)

利用这种方法计算多项式乘法的时间复杂度仍然为O(n2)

（选点O(n)，每次计算O(n)）

我们可以看到，两种方法的时间复杂度都为O(n2)，我们考虑对其进行优化

对于第一种方法，由于每个点的系数都是固定的，想要优化比较困难

对于第二种方法，貌似也没有什么好的优化方法，不过当你看完下面的知识，或许就不这么想了

## 复数

在介绍复数之前，首先介绍一些可能会用到的东西

### **向量**

同时具有大小和方向的量

在几何中通常用带有箭头的线段表示

### **圆的弧度制**

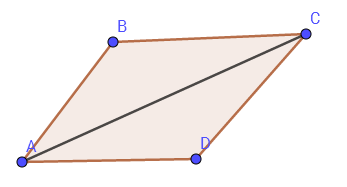
等于半径长的圆弧所对的[圆心角](https://baike.baidu.com/item/%E5%9C%86%E5%BF%83%E8%A7%92" \t "https://www.cnblogs.com/zwfymqz/p/_blank)叫做1[弧度](https://baike.baidu.com/item/%E5%BC%A7%E5%BA%A6" \t "https://www.cnblogs.com/zwfymqz/p/_blank)的角，用符号rad表示，读作弧度。用弧度作单位来度量角的制度叫做弧度制

公式:



180∘=πrad

### **平行四边形定则**



~~(好像画的不是很标准。。)~~

平行四边形定则：AB+AD=AC

### **复数**

#### **定义**

设a,b为实数，i2=−1，形如a+bi的数叫复数，其中i被称为虚数单位，复数域是目前已知最大的域

在复平面中，x代表实数，y轴（除原点外的点）代表虚数，从原点(0,0)到(a,b)的向量表示复数a+bi

模长：从原点(0,0)到点(a,b)的距离，即

幅角：假设以逆时针为正方向，从x轴正半轴到已知向量的转角的有向角叫做幅角

### **运算法则**

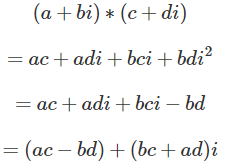
加法：

因为在复平面中，复数可以被表示为向量，因此复数的加法与向量的加法相同，都满足平行四边形定则（就是上面那个）

乘法：

几何定义：复数相乘，模长相乘，幅角相加

代数定义：



### **单位根**

下文中，默认n为2的正整数次幂

在复平面上，以原点为圆心，1为半径作圆，所得的圆叫****单位圆****。以圆点为起点，圆的n等分点为终点，做n个向量，设幅角为正且最小的向量对应的复数为ωn，称为n次单位根。

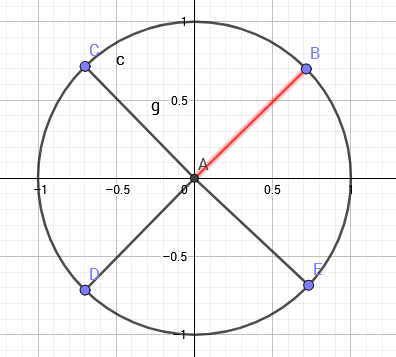
根据复数乘法的运算法则，其余n−1个复数为

注意（对应复平面上以x轴为正方向的向量）

那么如何计算它们的值呢？这个问题可以由欧拉公式解决



例如

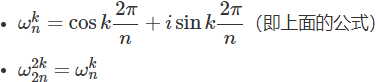


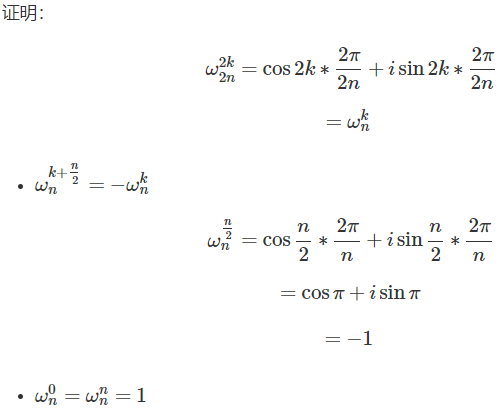
图中向量AB表示的复数为8次单位根

单位根的幅角为周角的1/n

在代数中，若zn=1，我们把z称为n次单位根

### **单位根的性质**





 讲了这么多，貌似跟我们的正题没啥关系啊。。

 OK！各位坐稳了，前方高能！

[回到顶部](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_labelTop)

## 快速傅里叶变换

我们前面提到过，一个n次多项式可以被n个点唯一确定。

那么我们可以把单位根的0到n−1次幂带入，这样也可以把这个多项式确定出来。但是这样仍然是O(n2)的呀！

我们设多项式A(x)的系数为(a0,a1,a2,…,an−1)

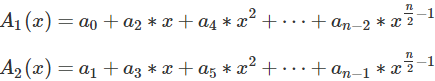
那么

A(x)=a0+a1\*x+a2\*x2+a3\*x3+a4\*x4+a5\*x5+⋯+an−2\*xn−2+an−1\*xn−1

将其下标按照奇偶性分类

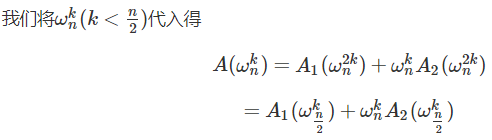
A(x)=(a0+a2\*x2+a4\*x4+⋯+an−2\*xn−2)+(a1\*x+a3\*x3+a5\*x5+⋯+an−1\*xn−1)

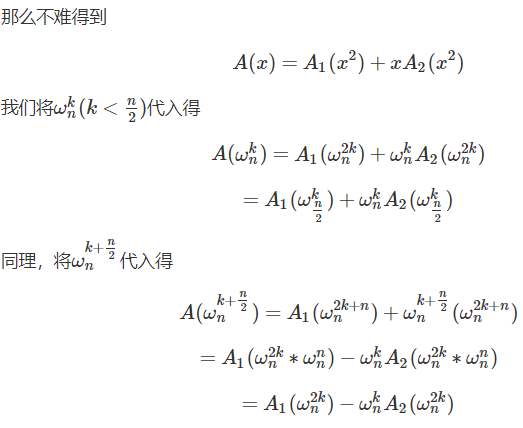
设



那么不难得到

A(x)=A1(x2)+xA2(x2)





大家有没有发现什么规律？

没错！这两个式子只有一个常数项不同！

那么当我们在枚举第一个式子的时候，我们可以O(1)的得到第二个式子的值



所以我们将原来的问题缩小了一半！

而缩小后的问题仍然满足原问题的性质，所以我们可以递归的去搞这件事情！

直到多项式仅剩一个常数项，这时候我们直接返回就好啦

时间复杂度：

不难看出FFT是类似于线段树一样的分治算法。

因此它的时间复杂度为O(nlogn)

[回到顶部](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_labelTop)

## 快速傅里叶逆变换

不要以为FFT到这里就结束了。

我们上面的讨论是基于点值表示法的。

但是在平常的学习和研究中很少用点值表示法来表示一个多项式。

所以我们要考虑如何把点值表示法转换为系数表示法，这个过程叫做****傅里叶逆变换****

(y0,y1,y2,…,yn−1)为(a0,a1,a2,…,an−1)的傅里叶变换（即点值表示）

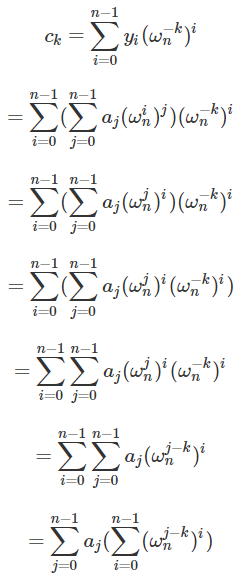
设有另一个向量(c0,c1,c2,…,cn−1)满足



即多项式B(x)=y0,y1x,y2x2,…,yn−1xn−1在处的点值表示

emmmm又到推公式时间啦

(c0,c1,c2,…,cn−1)满足



设

将代入得

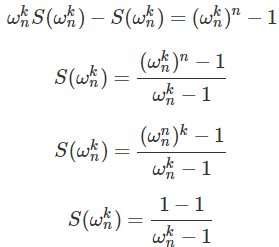


当k!=0时

等式两边同乘得



两式相减得



观察这个式子，不难看出它分母不为0，但是分子为0

因此，当K!=0时，

那当k=0时呢？

很显然，

继续考虑刚刚的式子

  
当j≠k时，值为0  
当j=k时，值为n  
因此，





这样我们就得到点值与系数之间的表示啦

## 理论总结

至此，FFT的基础理论部分就结束了。

我们来小结一下FFT是怎么成功实现的

首先，人们在用系数表示法研究多项式的时候遇阻

于是开始考虑能否用点值表示法优化这个东西。

然后根据复数的两条性质（这个思维跨度比较大）得到了一种分治算法。

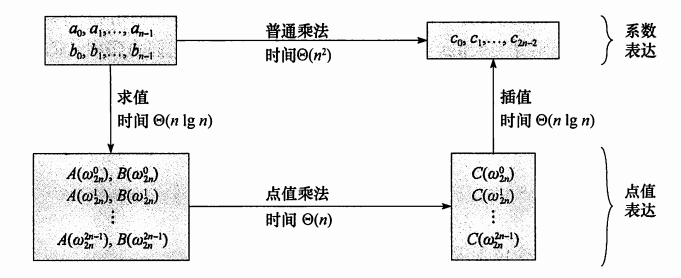
最后又推了一波公式，找到了点值表示法与系数表示法之间转换关系。

emmmm

其实FFT的实现思路大概就是

系数表示法—>点值表示法—>系数表示法

引用一下远航之曲大佬的图



当然，在实现的过程中还有很多技巧

我们根据代码来理解一下

[回到顶部](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_labelTop)

## 递归实现

递归实现的方法比较简单。

就是按找我们上面说的过程，不断把要求的序列分成两部分，再进行合并

在c++的STL中提供了现成的complex类，但是我不建议大家用，毕竟手写也就那么几行，而且万一某个毒瘤卡STL那岂不是很ＧＧ？

 递归版

// luogu-judger-enable-o2

#include<iostream>

#include<cstdio>

#include<cmath>

using namespace std;

const int MAXN = 4 \* 1e6 + 10;

inline int read() {

char c = getchar(); int x = 0, f = 1;

while (c < '0' || c > '9') {if (c == '-')f = -1; c = getchar();}

while (c >= '0' && c <= '9') {x = x \* 10 + c - '0'; c = getchar();}

return x \* f;

}

const double Pi = acos(-1.0);

struct complex {

double x, y;

complex (double xx = 0, double yy = 0) {x = xx, y = yy;}

} a[MAXN], b[MAXN];

complex operator + (complex a, complex b) { return complex(a.x + b.x , a.y + b.y);}

complex operator - (complex a, complex b) { return complex(a.x - b.x , a.y - b.y);}

complex operator \* (complex a, complex b) { return complex(a.x \* b.x - a.y \* b.y , a.x \* b.y + a.y \* b.x);} //不懂的看复数的运算那部分

void fast\_fast\_tle(int limit, complex \*a, int type) {

if (limit == 1) return ; //只有一个常数项

complex a1[limit >> 1], a2[limit >> 1];

for (int i = 0; i <= limit; i += 2) //根据下标的奇偶性分类

a1[i >> 1] = a[i], a2[i >> 1] = a[i + 1];

fast\_fast\_tle(limit >> 1, a1, type);

fast\_fast\_tle(limit >> 1, a2, type);

complex Wn = complex(cos(2.0 \* Pi / limit) , type \* sin(2.0 \* Pi / limit)), w = complex(1, 0);

//Wn为单位根，w表示幂

for (int i = 0; i < (limit >> 1); i++, w = w \* Wn) //这里的w相当于公式中的k

a[i] = a1[i] + w \* a2[i],

a[i + (limit >> 1)] = a1[i] - w \* a2[i]; //利用单位根的性质，O(1)得到另一部分

}

int main() {

int N = read(), M = read();

for (int i = 0; i <= N; i++) a[i].x = read();

for (int i = 0; i <= M; i++) b[i].x = read();

int limit = 1; while (limit <= N + M) limit <<= 1;

fast\_fast\_tle(limit, a, 1);

fast\_fast\_tle(limit, b, 1);

//后面的1表示要进行的变换是什么类型

//1表示从系数变为点值

//-1表示从点值变为系数

//至于为什么这样是对的，可以参考一下c向量的推导过程，

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

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

fast\_fast\_tle(limit, a, -1);

for (int i = 0; i <= N + M; i++) printf("%d ", (int)(a[i].x / limit + 0.5)); //按照我们推倒的公式，这里还要除以n

return 0;

}

递归版

update:递归版我本地是可以AC的，只要开大数组就可以了，但是交到洛谷上会WA0



woc? ~~脸好疼。。。。。。~~

咳咳。

速度什么的才不是关键呢？

关键是我们AC不了啊啊啊

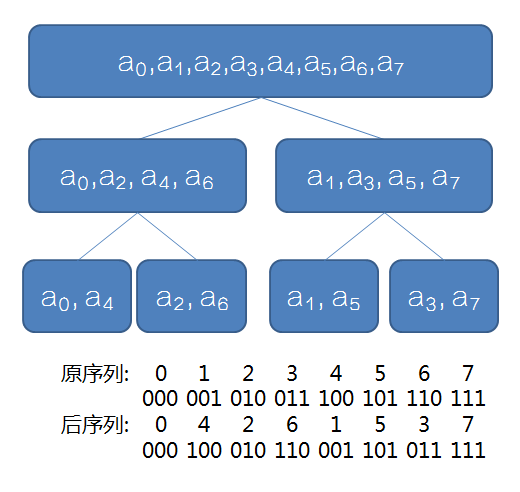
表着急，AC不了不代表咱们的算法不对，只能说这种实现方法太low了

下面介绍一种更高效的方法

[回到顶部](https://www.cnblogs.com/zwfymqz/p/8244902.html" \l "_labelTop)

## 迭代实现

~~再盗一下那位大佬的图~~



观察一下原序列和反转后的序列？

聪明的你有没有看出什么显而易见的性质？

没错！

我们需要求的序列实际是原序列下标的二进制反转！

因此我们对序列按照下标的奇偶性分类的过程其实是没有必要的

这样我们可以O(n)的利用某种操作得到我们要求的序列，然后不断向上合并就好了

// luogu-judger-enable-o2

#include<iostream>

#include<cstdio>

#include<cmath>

using namespace std;

const int MAXN = 1e7 + 10;

inline int read() {

char c = getchar(); int x = 0, f = 1;

while (c < '0' || c > '9') {if (c == '-')f = -1; c = getchar();}

while (c >= '0' && c <= '9') {x = x \* 10 + c - '0'; c = getchar();}

return x \* f;

}

const double Pi = acos(-1.0);

struct complex {

double x, y;

complex (double xx = 0, double yy = 0) {x = xx, y = yy;}

} a[MAXN], b[MAXN];

complex operator + (complex a, complex b) { return complex(a.x + b.x , a.y + b.y);}

complex operator - (complex a, complex b) { return complex(a.x - b.x , a.y - b.y);}

complex operator \* (complex a, complex b) { return complex(a.x \* b.x - a.y \* b.y , a.x \* b.y + a.y \* b.x);} //不懂的看复数的运算那部分

int N, M;

int l, r[MAXN];

int limit = 1;

void fast\_fast\_tle(complex \*A, int type) {

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

if (i < r[i]) swap(A[i], A[r[i]]); //求出要迭代的序列

for (int mid = 1; mid < limit; mid <<= 1) { //待合并区间的长度的一半

complex Wn( cos(Pi / mid) , type \* sin(Pi / mid) ); //单位根

for (int R = mid << 1, j = 0; j < limit; j += R) { //R是区间的长度，j表示前已经到哪个位置了

complex w(1, 0); //幂

for (int k = 0; k < mid; k++, w = w \* Wn) { //枚举左半部分

complex x = A[j + k], y = w \* A[j + mid + k]; //蝴蝶效应

A[j + k] = x + y;

A[j + mid + k] = x - y;

}

}

}

}

int main() {

int N = read(), M = read();

for (int i = 0; i <= N; i++) a[i].x = read();

for (int i = 0; i <= M; i++) b[i].x = read();

while (limit <= N + M) limit <<= 1, l++;

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

r[i] = ( r[i >> 1] >> 1 ) | ( (i & 1) << (l - 1) ) ;

// 在原序列中 i 与 i/2 的关系是 ： i可以看做是i/2的二进制上的每一位左移一位得来

// 那么在反转后的数组中就需要右移一位，同时特殊处理一下奇数

fast\_fast\_tle(a, 1);

fast\_fast\_tle(b, 1);

for (int i = 0; i <= limit; i++) a[i] = a[i] \* b[i];

fast\_fast\_tle(a, -1);

for (int i = 0; i <= N + M; i++)

printf("%d ", (int)(a[i].x / limit + 0.5));

return 0;

}

<https://www.cnblogs.com/zwfymqz/p/8244902.html>