卷积及其相关

ShichengXiao

雅礼中学

2018 年 8 月 22 日

引入

在我们走进卷积之前，先来了解一下两个多项式的卷积的定义

引入

在我们走进卷积之前，先来了解一下两个多项式的卷积的定义

定义两个多项式 $A(x)=\sum_{i=0}^{n-1}a_ix^i$ 与 $B(x)=\sum_{i=0}^{n-1}b_ix^i$，那么这两个多项式的卷积即为

$$C(x)=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}a_ib_jx^{i+j}$$

引入

在我们走进卷积之前，先来了解一下两个多项式的卷积的定义

定义两个多项式 $A(x)=\sum_{i=0}^{n-1}a_ix^i$ 与 $B(x)=\sum_{i=0}^{n-1}b_ix^i$，那么这两个多项式的卷积即为

$$C(x)=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}a_ib_jx^{i+j}$$

不难发现，如果使用暴力的话，时间复杂度为 $O(n^2)$ 的。利用我们今天所学的知识，我们就可以将其优化到 $O(n\log n)$。

引入

在我们走进卷积之前，先来了解一下两个多项式的卷积的定义

定义两个多项式 $A(x)=\sum_{i=0}^{n-1}a_ix^i$ 与 $B(x)=\sum_{i=0}^{n-1}b_ix^i$，那么这两个多项式的卷积即为

$$C(x)=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}a_ib_jx^{i+j}$$

不难发现，如果使用暴力的话，时间复杂度为 $O(n^2)$ 的。利用我们今天所学的知识，我们就可以将其优化到 $O(n\log n)$。

今天主要介绍的是 FFT/NTT 这两个求多项式卷积的算法，此外还会介绍一个求集合卷积的算法 FWT。

预备知识

首先来看一下 FFT。

预备知识

首先来看一下 FFT。

在介绍 FFT 之前，先来介绍一些预备的数学知识。

预备知识

首先来看一下 FFT。

在介绍 FFT 之前，先来介绍一些预备的数学知识。

预备的数学知识比较多，如果没有接触过 FFT 的话，可能会觉得有点恶心。

预备知识

首先来看一下 FFT。

在介绍 FFT 之前，先来介绍一些预备的数学知识。

预备的数学知识比较多，如果没有接触过 FFT 的话，可能会觉得有点恶心。

我会尽量讲解清楚，没学过的同学应该（?）能听懂，学过的同学也可以加深对于 FFT 的理解。

复数的相关知识

单位复数根：n 次单位复数根，指的是满足 $\omega^n = 1$ 的所有 ω。可以证明，n 次单位复数根有恰好 n 个，他们是 $e^{\frac{2k\pi i}{n}}$。其中 i 为虚数单位，$k \in [0, n-1]$ 且 $k \in \mathrm{Z}$。

复数的相关知识

单位复数根：n 次单位复数根，指的是满足 $\omega^n = 1$ 的所有 ω。可以证明，n 次单位复数根有恰好 n 个，他们是 $e^{\frac{2k\pi i}{n}}$。其中 i 为虚数单位，$k \in [0, n-1]$ 且 $k \in \mathrm{Z}$。

关于一个数的虚数次幂的定义，我们有欧拉公式（欧拉公式真多……）

$$e^{i\theta} = \cos\theta + i\sin\theta$$

复数的相关知识

单位复数根：n 次单位复数根，指的是满足 $\omega^n = 1$ 的所有 ω。可以证明，n 次单位复数根有恰好 n 个，他们是 $e^{\frac{2k\pi i}{n}}$。其中 i 为虚数单位，$k \in [0, n-1]$ 且 $k \in \mathrm{Z}$。

关于一个数的虚数次幂的定义，我们有欧拉公式（欧拉公式真多……）

$$e^{i\theta} = \cos\theta + i\sin\theta$$

其中 $\omega_n = e^{\frac{2i\pi}{n}}$ 被称为主 n 次单位根。我们下文所用的所有的 ω_n 都是这个定义。

复数的相关知识

消去引理：对于任意整数 $n \geq 0, k \geq 0, d \geq 0$，有 $\omega_{dn}^{dk} = \omega_n^k$。

复数的相关知识

消去引理：对于任意整数 $n \geq 0, k \geq 0, d \geq 0$，有 $\omega_{dn}^{dk} = \omega_n^k$。

至于证明，可以直接将上文中的复数根的计算方式带入化简即可，大家感兴趣可以自己推一推。

复数的相关知识

消去引理：对于任意整数 $n \geq 0, k \geq 0, d \geq 0$，有 $\omega_{dn}^{dk} = \omega_n^k$。

至于证明，可以直接将上文中的复数根的计算方式带入化简即可，大家感兴趣可以自己推一推。

一个推论：对任意正偶数 n，有 $\omega_n^{\frac{n}{2}} = \omega_2 = -1$。

复数的相关知识

折半引理：$\omega_n^{k+\frac{n}{2}} = -\omega_n^k$

复数的相关知识

折半引理：$\omega_n^{k+\frac{n}{2}} = -\omega_n^k$

证明比较简单，$\omega_n^{k+\frac{n}{2}} = \omega_n^k \times \omega_n^{\frac{n}{2}} = -\omega_n^k$

复数的相关知识

求和引理：当 $k > 0$ 时，有 $\sum_{i=0}^{n-1} (\omega_n^k)^i = 0$

复数的相关知识

求和引理：当 $k > 0$ 时，有 $\sum_{i=0}^{n-1} (\omega_n^k)^i = 0$

证明可以考虑利用等比数列求和，错位相减的方法。令上式为 S，两边同乘 ω_n^k 得

$$\omega_n^k S = \sum_{i=1}^{n} (\omega_n^k)^i$$

复数的相关知识

求和引理：当 $k > 0$ 时，有 $\sum_{i=0}^{n-1} (\omega_n^k)^i = 0$

证明可以考虑利用等比数列求和，错位相减的方法。令上式为 S，两边同乘 ω_n^k 得

$$\omega_n^k S = \sum_{i=1}^{n} (\omega_n^k)^i$$

两式相减，整理后可得：

$$S = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = 0$$

得证。

多项式的表示方法

对于一个多项式，我们一般使用的表示方法，就是用一些系数来表示一个多项式。相信大家很熟悉。

多项式的表示方法

对于一个多项式，我们一般使用的表示方法，就是用一些系数来表示一个多项式。相信大家很熟悉。

但是除了系数表示法之外，我们还有一种常用的表示方式，即为点值表示法。

多项式的表示方法

对于一个多项式，我们一般使用的表示方法，就是用一些系数来表示一个多项式。相信大家很熟悉。

但是除了系数表示法之外，我们还有一种常用的表示方式，即为点值表示法。

点值表示法，顾名思义，就是用一些点的函数值来表示这个多项式。

多项式的表示方法

对于一个多项式，我们一般使用的表示方法，就是用一些系数来表示一个多项式。相信大家很熟悉。

但是除了系数表示法之外，我们还有一种常用的表示方式，即为点值表示法。

点值表示法，顾名思义，就是用一些点的函数值来表示这个多项式。

容易证明，对于一个 n 次多项式，只需要 n 个点值就可以表示这个多项式。

算法过程

基础的数学知识大概就到这里，接下来我们看一下 FFT 的算法过程。

算法过程

基础的数学知识大概就到这里，接下来我们看一下 FFT 的算法过程。

由我们刚刚所讲的点值表示法，不难发现，如果我们得到了两个多项式在相同点的点值表达形式，那么这两个多项式的卷积所得到的多项式的点值，可以在 $O(n)$ 的复杂度内轻松得到。

算法过程

基础的数学知识大概就到这里，接下来我们看一下 FFT 的算法过程。

由我们刚刚所讲的点值表示法，不难发现，如果我们得到了两个多项式在相同点的点值表达形式，那么这两个多项式的卷积所得到的多项式的点值，可以在 $O(n)$ 的复杂度内轻松得到。

于是，整个算法的雏形也就出来了：

算法过程

基础的数学知识大概就到这里，接下来我们看一下 FFT 的算法过程。

由我们刚刚所讲的点值表示法，不难发现，如果我们得到了两个多项式在相同点的点值表达形式，那么这两个多项式的卷积所得到的多项式的点值，可以在 $O(n)$ 的复杂度内轻松得到。

于是，整个算法的雏形也就出来了：

- 将两个多项式转化为点值的形式
- 将这两个多项式的点值形式相乘
- 将所得的点值形式逆变换为多项式的形式

算法过程

上述步骤中，第二步实现起来很轻松，于是在 $O(n \log n)$ 的时间内实现第一步和第三步也就成为了整个算法的重点。

算法过程

上述步骤中，第二步实现起来很轻松，于是在 $O(n\log n)$ 的时间内实现第一步和第三步也就成为了整个算法的重点。

利用复数的性质，我们希望求出的点值，是在 $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ 下的点值。

算法过程

上述步骤中，第二步实现起来很轻松，于是在 $O(n \log n)$ 的时间内实现第一步和第三步也就成为了整个算法的重点。

利用复数的性质，我们希望求出的点值，是在 $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ 下的点值。

这个过程被称为离散傅里叶变换。

离散傅里叶变换

考虑对于一个多项式，按照其下标的奇偶性分为两个部分，即：

$$A(x) = (a_0 + a_2 x^2 + a_4 x^4 + \cdots + a_{n-2} x^{n-2}) + (a_1 x + a_3 x^3 + \cdots + a_{n-1} x^{n-1})$$

离散傅里叶变换

考虑对于一个多项式，按照其下标的奇偶性分为两个部分，即：

$$A(x) = (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) + (a_1x + a_3x^3 + \cdots + a_{n-1}x^{n-1})$$

令

$$A_1(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{\frac{n}{2}-1}$$
$$A_2(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{\frac{n}{2}-1}$$

离散傅里叶变换

考虑对于一个多项式，按照其下标的奇偶性分为两个部分，即：

$$A(x) = (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) + (a_1x + a_3x^3 + \cdots + a_{n-1}x^{n-1})$$

令

$$A_1(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{\frac{n}{2}-1}$$
$$A_2(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{\frac{n}{2}-1}$$

则有

$$A(x) = A_1(x^2) + xA_2(x^2)$$

离散傅里叶变换

假设我们现在要求 $A(\omega_n^k)$ $(k \in [0, n-1])$：

离散傅里叶变换

假设我们现在要求 $A(\omega_n^k)\ (k \in [0, n-1])$:

考虑 $A(\omega_n^k)$，$k \in [0, \frac{n}{2} - 1]$

$$\begin{aligned} A(\omega_n^k) &= A_1(\omega_n^{2k}) + \omega_n^k A_2(\omega_n^{2k}) \\ &= A_1(\omega_{\frac{n}{2}}^k) + \omega_n^k A_2(\omega_{\frac{n}{2}}^k) \end{aligned}$$

离散傅里叶变换

假设我们现在要求 $A(\omega_n^k)\ (k \in [0, n-1])$:

考虑 $A(\omega_n^k)$，$k \in [0, \frac{n}{2}-1]$

$$\begin{aligned} A(\omega_n^k) &= A_1(\omega_n^{2k}) + \omega_n^k A_2(\omega_n^{2k}) \\ &= A_1(\omega_{\frac{n}{2}}^k) + \omega_n^k A_2(\omega_{\frac{n}{2}}^k) \end{aligned}$$

考虑 $A(\omega_n^{k+\frac{n}{2}})$，$k \in [0, \frac{n}{2}-1]$

$$\begin{aligned} A(\omega_n^{k+\frac{n}{2}}) &= A_1(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} A_2(\omega_n^{2k+n}) \\ &= A_1(\omega_n^{2k} \times \omega_n^n) - \omega_n^k A_2(\omega_n^{2k} \times \omega_n^n) \\ &= A_1(\omega_n^{2k}) - \omega_n^k A_2(\omega_n^{2k}) = A_1(\omega_{\frac{n}{2}}^k) - \omega_n^k A_2(\omega_{\frac{n}{2}}^k) \end{aligned}$$

离散傅里叶变换

到这里我们终于发现，只要我们求出了 $A_1(x)$ 和 $A_2(x)$ 在 $\omega_{\frac{n}{2}}^{k}(k \in [0, \frac{n}{2}-1]$ 且 $k \in \mathrm{Z})$ 下的点值，我们就可以在 $O(n)$ 的时间复杂度内，求出 $A(x)$ 在 $\omega_n^k(k \in [0, n-1]$ 且 $k \in \mathrm{Z})$ 下的点值。

离散傅里叶变换

到这里我们终于发现，只要我们求出了 $A_1(x)$ 和 $A_2(x)$ 在 $\omega_{\frac{n}{2}}^{k}(k \in [0, \frac{n}{2}-1]$ 且 $k \in \mathrm{Z})$ 下的点值，我们就可以在 $O(n)$ 的时间复杂度内，求出 $A(x)$ 在 $\omega_n^k(k \in [0, n-1]$ 且 $k \in \mathrm{Z})$ 下的点值。

注意到将 a_i 重新排列一下并递归求解，就可以求出 $A_1(x)$ 和 $A_2(x)$ 的点值，因此我们可以递归实现这个过程。

考虑这个算法的时间复杂度，$T(n) = 2\,T(\frac{n}{2}) + O(n)$。

离散傅里叶变换

到这里我们终于发现，只要我们求出了 $A_1(x)$ 和 $A_2(x)$ 在 $\omega_{\frac{n}{2}}^{k}(k \in [0, \frac{n}{2}-1]$ 且 $k \in \mathrm{Z})$ 下的点值，我们就可以在 $O(n)$ 的时间复杂度内，求出 $A(x)$ 在 $\omega_n^k(k \in [0, n-1]$ 且 $k \in \mathrm{Z})$ 下的点值。

注意到将 a_i 重新排列一下并递归求解，就可以求出 $A_1(x)$ 和 $A_2(x)$ 的点值，因此我们可以递归实现这个过程。

考虑这个算法的时间复杂度，$T(n) = 2\,T(\frac{n}{2}) + O(n)$。

根据主定理，这个阶段算法的时间复杂度为 $O(n \log n)$。

离散傅里叶逆变换

但是光有离散傅里叶变换还不够，上文已经说过，除了要能从多项式快速转化为点值形式之外，我们还要能快速地实现逆变换。

离散傅里叶逆变换

但是光有离散傅里叶变换还不够，上文已经说过，除了要能从多项式快速转化为点值形式之外，我们还要能快速地实现逆变换。

此处为了方便，我们先给出结论，再来讲讲证明的过程。

离散傅里叶逆变换

但是光有离散傅里叶变换还不够，上文已经说过，除了要能从多项式快速转化为点值形式之外，我们还要能快速地实现逆变换。

此处为了方便，我们先给出结论，再来讲讲证明的过程。

使用单位根的倒数代替单位根，做一次类似快速傅里叶变换的过程，再将结果每个数除以 n，即为傅里叶逆变换的结果。

离散傅里叶逆变换

设 y_i 为多项式 $\sum_{j=0}^{n-1} a_j x^j$ 在 ω_n^i 下的点值，即已经经过了离散傅里叶变换之后的结果。

令多项式 $B(x) = \sum_{i=0}^{n-1} y_i x^i$，考虑多项式 $B(x)$ 在 ω_n^{-k} $(k \in [0, n-1])$ 的点值，记作 c_k，即

$$c_k = \sum_{i=0}^{n-1} y_i (\omega_n^{-k})^i$$

离散傅里叶逆变换

然后是枯燥的化式子：

离散傅里叶逆变换

然后是枯燥的化式子：

$$
\begin{aligned}
c_k &= \sum_{i=0}^{n-1} y_i(\omega_n^{-k})^i \\
&= \sum_{i=0}^{n-1}(\sum_{j=0}^{n-1} a_j(\omega_n^i)^j)(\omega_n^{-k})^i \\
&= \sum_{i=0}^{n-1}(\sum_{j=0}^{n-1} a_j(\omega_n^j)^i)(\omega_n^{-k})^i \\
&= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_j(\omega_n^{j-k})^i = \sum_{j=0}^{n-1} a_j(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)
\end{aligned}
$$

离散傅里叶逆变换

现在式子化成了这个样子：$\sum_{j=0}^{n-1} a_j(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)$

离散傅里叶逆变换

现在式子化成了这个样子：$\sum_{j=0}^{n-1} a_j(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)$

由我们之前所说的求和引理，当 $j=k$ 时，$(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)=n$，否则为 0。

离散傅里叶逆变换

现在式子化成了这个样子：$\sum_{j=0}^{n-1} a_j(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)$

由我们之前所说的求和引理，当 $j=k$ 时，$(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)=n$，否则为 0。

所以我们有 $c_i=na_i$，即 $a_i=\frac{1}{n}c_i$，得证。

离散傅里叶逆变换

现在式子化成了这个样子：$\sum_{j=0}^{n-1} a_j(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)$

由我们之前所说的求和引理，当 $j=k$ 时，$(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)=n$，否则为 0。

所以我们有 $c_i=na_i$，即 $a_i=\frac{1}{n}c_i$，得证。

到此，整个算法过程的各种证明终于结束了。不难发现，其总时间复杂度同样为 $O(n\log n)$。

代码实现

代码实现上，一种方法是直接按照之前算法所介绍的，直接递归求解。

代码实现

代码实现上，一种方法是直接按照之前算法所介绍的，直接递归求解。

每次递归求解时，先把数组按照下标 $0, 2, 4, \ldots, n-2, 1, 3, 5, \ldots, n-1$ 的顺序重新排列一下，再分别递归求解并合并答案。

代码实现

代码实现上，一种方法是直接按照之前算法所介绍的，直接递归求解。

每次递归求解时，先把数组按照下标 $0, 2, 4, \ldots, n-2, 1, 3, 5, \ldots, n-1$ 的顺序重新排列一下，再分别递归求解并合并答案。

但由于递归求解的运行效率并不理想，所以这里不介绍这种递归做法，而是另一种迭代做法。

代码实现

代码实现上，一种方法是直接按照之前算法所介绍的，直接递归求解。

每次递归求解时，先把数组按照下标 $0, 2, 4, \ldots, n-2, 1, 3, 5, \ldots, n-1$ 的顺序重新排列一下，再分别递归求解并合并答案。

但由于递归求解的运行效率并不理想，所以这里不介绍这种递归做法，而是另一种迭代做法。

然而这种迭代做法要讲清楚似乎不是那么简单.....

二进制位翻转

首先要跳出递归求解，就必须要把求解顺序的规律做一下研究。

二进制位翻转

首先要跳出递归求解，就必须要把求解顺序的规律做一下研究。

数的编号	编号对应二进制	分治顺序	分治顺序对应二进制
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

二进制位翻转

不难发现，分治顺序对应二进制就是编号对应二进制位翻转的结果。

二进制位翻转

不难发现，分治顺序对应二进制就是编号对应二进制位翻转的结果。

代码实现这个过程，可以利用之前所求的结果，代码十分简洁。

蝴蝶操作

考虑我们在合并答案的过程中，$A_1(\omega_{\frac{n}{2}}^k)$ 和 $A_2(\omega_{\frac{n}{2}}^k)$ 的值，将分别被存放在 $a[k]$ 和 $a[k+\frac{n}{2}]$ 中。

蝴蝶操作

考虑我们在合并答案的过程中，$A_1(\omega_{\frac{n}{2}}^k)$ 和 $A_2(\omega_{\frac{n}{2}}^k)$ 的值，将分别被存放在 $a[k]$ 和 $a[k+\frac{n}{2}]$ 中。

而 $A(\omega_n^k)$ 和 $A(\omega_n^{k+\frac{n}{2}})$ 将会被存放在 $a[k]$ 和 $a[k+\frac{n}{2}]$ 中。因此我们可以这样来实现这个过程：

蝴蝶操作

考虑我们在合并答案的过程中，$A_1(\omega_{\frac{n}{2}}^k)$ 和 $A_2(\omega_{\frac{n}{2}}^k)$ 的值，将分别被存放在 $a[k]$ 和 $a[k+\frac{n}{2}]$ 中。

而 $A(\omega_n^k)$ 和 $A(\omega_n^{k+\frac{n}{2}})$ 将会被存放在 $a[k]$ 和 $a[k+\frac{n}{2}]$ 中。因此我们可以这样来实现这个过程：

$$u = a(k), v = \omega_n^k \times a(\frac{n}{2} + k);$$
$$a(k) = u + v, a(\frac{n}{2} + k) = u - v;$$

其他实现细节

注意到整个分治的过程，都要求 n 为 2 的整数次幂。如果题目中的 n 不是 2 的整数次幂，我们需要用 0 将其补齐。

其他实现细节

注意到整个分治的过程，都要求 n 为 2 的整数次幂。如果题目中的 n 不是 2 的整数次幂，我们需要用 0 将其补齐。

关于复数的计算，可以用系统自带的复数类 $\mathrm{complex}$，不过自己手写的效率会高一点。

其他实现细节

注意到整个分治的过程，都要求 n 为 2 的整数次幂。如果题目中的 n 不是 2 的整数次幂，我们需要用 0 将其补齐。

关于复数的计算，可以用系统自带的复数类 $\mathrm{complex}$，不过自己手写的效率会高一点。

至此，FFT 终终终终终于讲完了。

从 FFT 到 NTT

FFT 虽然好，但是有一个问题，就是其精度误差，这意味着 FFT 对于每一项过大的多项式，卷起来的精度误差也会非常大。一般来讲，FFT 最极限允许的每一项不超过 32768 左右。当题目要求将答案对某个数取模时，这意味着每一项都是 10^9 级别的数，问题就比较麻烦了。于是我们有 NTT 来解决这个问题。

从 FFT 到 NTT

FFT 虽然好，但是有一个问题，就是其精度误差，这意味着 FFT 对于每一项过大的多项式，卷起来的精度误差也会非常大。一般来讲，FFT 最极限允许的每一项不超过 32768 左右。当题目要求将答案对某个数取模时，这意味着每一项都是 10^9 级别的数，问题就比较麻烦了。于是我们有 NTT 来解决这个问题。

与 FFT 所利用的复数性质不同，NTT 利用的是模意义下的一些性质。事实上，我们只要能够证明复数中的那些引理在这样一个模意义下的系统上仍然成立，NTT 的性质也就很显然了。

从 FFT 到 NTT

FFT 虽然好，但是有一个问题，就是其精度误差，这意味着 FFT 对于每一项过大的多项式，卷起来的精度误差也会非常大。一般来讲，FFT 最极限允许的每一项不超过 32768 左右。当题目要求将答案对某个数取模时，这意味着每一项都是 10^9 级别的数，问题就比较麻烦了。于是我们有 NTT 来解决这个问题。

与 FFT 所利用的复数性质不同，NTT 利用的是模意义下的一些性质。事实上，我们只要能够证明复数中的那些引理在这样一个模意义下的系统上仍然成立，NTT 的性质也就很显然了。

在 NTT 中，我们用 $g_n = g^{\frac{p-1}{n}} \bmod p$ 来代替 ω_n（这里 g 为 p 的原根）。可以证明，原根的性质与单位复数根的许多性质都是相同的，因此直接做就好了。

一些实现上的细节

注意到根据我们上文的描述，$p-1$ 必须要是 n 的倍数，而 n 又是 2 的整数次幂，因此 NTT 对于模数的要求很高。常见的 NTT 模数主要有 998244353 或 1004535809 等。

一些实现上的细节

注意到根据我们上文的描述，$p-1$ 必须要是 n 的倍数，而 n 又是 2 的整数次幂，因此 NTT 对于模数的要求很高。常见的 NTT 模数主要有 998244353 或 1004535809 等。

如果模数不满足这种性质该怎么办呢？可以用 matthew99 在 2016 年论文中所讲的东西，大体思路是将每个数拆成一个两位的 32768 进制数，然后用 FFT 的精度去强行扛，对于这两位暴力分解计算。大家有兴趣的还可以去看看论文了解一下，论文中还有一些更加优秀的优化。

一些实现上的细节

注意到根据我们上文的描述，$p-1$ 必须要是 n 的倍数，而 n 又是 2 的整数次幂，因此 NTT 对于模数的要求很高。常见的 NTT 模数主要有 998244353 或 1004535809 等。

如果模数不满足这种性质该怎么办呢？可以用 matthew99 在 2016 年论文中所讲的东西，大体思路是将每个数拆成一个两位的 32768 进制数，然后用 FFT 的精度去强行扛，对于这两位暴力分解计算。大家有兴趣的还可以去看看论文了解一下，论文中还有一些更加优秀的优化。

实现上，预处理出所有的 g_i 可以在一定程度上提高效率。

例题

相信大家都听懂了上面的内容，没听懂也没关系，只要暂时知道了 FFT/NTT 能干啥，接下来的内容仍然能理解。

例题

相信大家都听懂了上面的内容，没听懂也没关系，只要暂时知道了 FFT/NTT 能干啥，接下来的内容仍然能理解。

题目都很简单，欢迎大家上来秒题。

例题

相信大家都听懂了上面的内容，没听懂也没关系，只要暂时知道了 FFT/NTT 能干啥，接下来的内容仍然能理解。

题目都很简单，欢迎大家上来秒题。可能没有那种让人直呼巧妙的题，抱歉让大家失望了。

BZOJ2194 快速傅立叶之二

请计算 $C_k = \sum_{i=k}^{n-1} a_i \times b_{i-k}$。$a, b$ 中的元素均为小于等于 100 的非负整数。数组下标从 0 开始。

BZOJ2194 快速傅立叶之二

请计算 $C_k = \sum_{i=k}^{n-1} a_i \times b_{i-k}$。$a, b$ 中的元素均为小于等于 100 的非负整数。数组下标从 0 开始。

$n \leq 10^5$

BZOJ2194 Solution

题目名字带来一种模板题的既视感，事实上这也确实是道模板题。

BZOJ2194 Solution

题目名字带来一种模板题的既视感，事实上这也确实是道模板题。

只需要将 b 数组翻转一下，就可以直接 FFT 了。

BZOJ2194 Solution

题目名字带来一种模板题的既视感，事实上这也确实是道模板题。

只需要将 b 数组翻转一下，就可以直接 FFT 了。

事实上，将某个数组翻转，也是 FFT 中的常见套路。

BZOJ4827 [HNOI2017] 礼物

给定两个长度为 n 的数组 a, b。你可以对任意一个数组做以下两种操作若干次：

BZOJ4827 [HNOI2017] 礼物

给定两个长度为 n 的数组 a, b。你可以对任意一个数组做以下两种操作若干次：

将数组中的每个数加上一个数 c；

将数组中的数字轮换一次，即新数组中第 i 个数，为原数组中的第 $i-1$ 个数，新数组中第 1 个数，为原数组中的第 n 个数。

BZOJ4827 [HNOI2017] 礼物

给定两个长度为 n 的数组 a, b。你可以对任意一个数组做以下两种操作若干次：

将数组中的每个数加上一个数 c；

将数组中的数字轮换一次，即新数组中第 i 个数，为原数组中的第 $i-1$ 个数，新数组中第 1 个数，为原数组中的第 n 个数。

经过操作后，你要最小化 $\sum_{i=1}^{n}(a_i - b_i)^2$ 的值。

BZOJ4827 [HNOI2017] 礼物

给定两个长度为 n 的数组 a, b。你可以对任意一个数组做以下两种操作若干次：

将数组中的每个数加上一个数 c；

将数组中的数字轮换一次，即新数组中第 i 个数，为原数组中的第 $i-1$ 个数，新数组中第 1 个数，为原数组中的第 n 个数。

经过操作后，你要最小化 $\sum_{i=1}^{n}(a_i - b_i)^2$ 的值。

$n \leq 50000$，每个数的值域 $m \leq 100$。

BZOJ4827 Solution

首先，对于 c，我们是可以不用枚举的。

BZOJ4827 Solution

首先，对于 c，我们是可以不用枚举的。

设 $f=\sum_{i=1}^{n}(a_i-b_i)^2$，$f'=\sum_{i=1}^{n}(a_i+c-b_i)^2$，则

$$f'-f=\sum_{i=1}^{n}(2a_i-2b_i+c)\times c=(2sumx-2sumy+c)\times c$$

BZOJ4827 Solution

首先，对于 c，我们是可以不用枚举的。

设 $f=\sum_{i=1}^{n}(a_i-b_i)^2$，$f'=\sum_{i=1}^{n}(a_i+c-b_i)^2$，则

$$f'-f=\sum_{i=1}^{n}(2a_i-2b_i+c)\times c=(2sumx-2sumy+c)\times c$$

注意到这是一个仅和 c 有关的二次函数，且与旋转操作无关系。当 $c=\frac{sumy-sumx}{n}$ 时答案最优。

BZOJ4827 Solution

再来考虑旋转操作。设将序列 a 翻转 k 次的答案为 f_k，不难把式子的平方项拆开，得到

$$f_k = \sum_{i=1}^{n}(a_{i+k} - b_i)^2 = \sum_{i=1}^{n} a_{i+k}^2 + b_i^2 - 2\sum_{i=1}^{n} a_{i+k}b_i$$

BZOJ4827 Solution

再来考虑旋转操作。设将序列 a 翻转 k 次的答案为 f_k，不难把式子的平方项拆开，得到

$$f_k = \sum_{i=1}^{n}(a_{i+k} - b_i)^2 = \sum_{i=1}^{n} a_{i+k}^2 + b_i^2 - 2\sum_{i=1}^{n} a_{i+k}b_i$$

这里我们对 a, b 数组都倍增了一次。

BZOJ4827 Solution

再来考虑旋转操作。设将序列 a 翻转 k 次的答案为 f_k，不难把式子的平方项拆开，得到

$$f_k = \sum_{i=1}^{n}(a_{i+k} - b_i)^2 = \sum_{i=1}^{n} a_{i+k}^2 + b_i^2 - 2\sum_{i=1}^{n} a_{i+k}b_i$$

这里我们对 a, b 数组都倍增了一次。

我们可以发现，前面两项的值是固定的，后面一项与我们刚刚所讲的 BZOJ2194 是完全一样的，于是直接将序列翻转并 FFT 就好了。

LOJ6388 [THUPC2018] 赛艇

给定一个 $n \times m$ 的 $0/1$ 矩阵。一个人在这个矩阵中走了 k 步，每一步都往上下左右中的一个走了一步。给定这个人每一步走的方向。已知这个人经过的每一步都没有经过 $0/1$ 矩阵中为 1 的位置。问合法的起点有多少种？保证至少有一组解。

LOJ6388 [THUPC2018] 赛艇

给定一个 $n \times m$ 的 $0/1$ 矩阵。一个人在这个矩阵中走了 k 步，每一步都往上下左右中的一个走了一步。给定这个人每一步走的方向。已知这个人经过的每一步都没有经过 $0/1$ 矩阵中为 1 的位置。问合法的起点有多少种？保证至少有一组解。

$n, m \leq 1500, k \leq 5 \times 10^6$，时间限制 $12s$。

LOJ6388 Solution

感觉这道题还是挺有意思的。FFT 的模型转化不太好想。

LOJ6388 Solution

感觉这道题还是挺有意思的。FFT 的模型转化不太好想。

我们考虑把这个人的轨迹也化成一个 $0/1$ 矩阵，其中 1 表示这个人经过的地方，0 表示没经过的地方，且这个矩阵的大小不大于地图矩阵。那么题目所求就可以转化为，这样两个矩阵匹配，不存在一个位置满足两个矩阵同时为 1。

LOJ6388 Solution

感觉这道题还是挺有意思的。FFT 的模型转化不太好想。

我们考虑把这个人的轨迹也化成一个 $0/1$ 矩阵，其中 1 表示这个人经过的地方，0 表示没经过的地方，且这个矩阵的大小不大于地图矩阵。那么题目所求就可以转化为，这样两个矩阵匹配，不存在一个位置满足两个矩阵同时为 1。

这个时候我们把两个矩阵都转化为序列，把其中一个序列翻转一下，然后做一遍 FFT。如果这个位置上的答案为 0，那么这就是一个可行方案。于是我们就在 $O(nm \log n \log m)$ 内解决了。

FFT 解决部分字符串问题

FFT 除了一些式子的计算之外，还可以解决一些字符串匹配问题，这也是 FFT 的一个经典套路。

Codeforces 954I Yet Another String Matching Problem

给定两个字符串 S 和 T ($|T| \leq |S|$)，询问 S 的 $|S| - |T| + 1$ 个长度为 $|T|$ 的子串与 T 这个串的距离。两个串的距离定义为，要使得两个串变为完全相同，每次可以把两个串中的某一个字符全部改为另一个字符的最小操作次数。$|T| \leq |S| \leq 125000$，字符集大小为 6。

Codeforces 954I Solution

首先可以发现，对于两个字符串，他们的距离可以这样计算：依次判断两个串对应位置位置，若前一个串第 i 个位置为 a_i，后一个串第 i 个位置为 b_i，那么我们就将 a_i 和 b_i 在并查集上连边。最后联通块的个数就是最后的答案。

Codeforces 954I Solution

首先可以发现，对于两个字符串，他们的距离可以这样计算：依次判断两个串对应位置位置，若前一个串第 i 个位置为 a_i，后一个串第 i 个位置为 b_i，那么我们就将 a_i 和 b_i 在并查集上连边。最后联通块的个数就是最后的答案。

要想快速判断对应位置上是否相等，有一个更快的方法，我们可以把 T 串倒过来，再对于每一种字母的对应情况，都做一遍 FFT 就可以了。最后再在并查集上连边判断即可。

Codeforces 954I Solution

首先可以发现，对于两个字符串，他们的距离可以这样计算：依次判断两个串对应位置位置，若前一个串第 i 个位置为 a_i，后一个串第 i 个位置为 b_i，那么我们就将 a_i 和 b_i 在并查集上连边。最后联通块的个数就是最后的答案。

要想快速判断对应位置上是否相等，有一个更快的方法，我们可以把 T 串倒过来，再对于每一种字母的对应情况，都做一遍 FFT 就可以了。最后再在并查集上连边判断即可。

时间复杂度 $O(36n\log n)$。

分治 FFT

利用 CDQ 分治的思想，FFT 可以求解形如 $f_n = \sum_{i=1}^{n-1} f_i \times a_{n-i}$ 的递推式。

分治 FFT

利用 CDQ 分治的思想，FFT 可以求解形如 $f_n = \sum_{i=1}^{n-1} f_i \times a_{n-i}$ 的递推式。

具体来说，在分治时，先计算 $f_{l\to mid}$ 的答案，然后将 $f_{l\to mid}$ 与 $a_{1\to r-l}$ 做卷积运算加到 $f_{mid+1\to r}$ 中。

分治 FFT

利用 CDQ 分治的思想，FFT 可以求解形如 $f_n = \sum_{i=1}^{n-1} f_i \times a_{n-i}$ 的递推式。

具体来说，在分治时，先计算 $f_{l\to mid}$ 的答案，然后将 $f_{l\to mid}$ 与 $a_{1\to r-l}$ 做卷积运算加到 $f_{mid+1\to r}$ 中。

时间复杂度会来到 $O(n\log^2 n)$。

BZOJ3456 城市规划

求 n 个点的简单无向连通图的个数。

BZOJ3456 城市规划

求 n 个点的简单无向连通图的个数。

$n \leq 130000$

BZOJ3456 Solution

像这种图计数问题，基本的套路就是反面计数。即用所有的方案减去不连通的图的方案数。

BZOJ3456 Solution

像这种图计数问题，基本的套路就是反面计数。即用所有的方案减去不连通的图的方案数。

这里也是一样的。枚举 1 号点所在的联通块的大小，强制让这个联通块联通，其余的边随便连。所以答案为：

BZOJ3456 Solution

像这种图计数问题，基本的套路就是反面计数。即用所有的方案减去不连通的图的方案数。

这里也是一样的。枚举 1 号点所在的联通块的大小，强制让这个联通块联通，其余的边随便连。所以答案为：

$$f_n = 2^{\binom{n}{2}} - \sum_{i=1}^{n-1} 2^{\binom{n-i}{2}} \times \binom{n-1}{i-1} \times f_i$$

BZOJ3456 Solution

像这种图计数问题，基本的套路就是反面计数。即用所有的方案减去不连通的图的方案数。

这里也是一样的。枚举 1 号点所在的联通块的大小，强制让这个联通块联通，其余的边随便连。所以答案为：

$$f_n = 2^{\binom{n}{2}} - \sum_{i=1}^{n-1} 2^{\binom{n-i}{2}} \times \binom{n-1}{i-1} \times f_i$$

然后可以转化为卷积形式，分治 FFT 即可。

Codeforces 553E Kyoya and Train

给出一幅 n 个点 m 条边的有向图，并给出参数 T，你要从 1 号点走到 n 号点。经过每一条边都要花费时间和金钱，第 i 条边需要花费 $cost_i$ 的金钱，并且经过该边花费时间为 t 的概率是 $p_{i,t}, 1 \leq t \leq T$。如果最后你花费的总时间大于 T，你就要付出额外 x 的金钱。请最小化期望花费的金钱。

Codeforces 553E Kyoya and Train

给出一幅 n 个点 m 条边的有向图，并给出参数 T，你要从 1 号点走到 n 号点。经过每一条边都要花费时间和金钱，第 i 条边需要花费 $cost_i$ 的金钱，并且经过该边花费时间为 t 的概率是 $p_{i,t}, 1 \leq t \leq T$。如果最后你花费的总时间大于 T，你就要付出额外 x 的金钱。请最小化期望花费的金钱。

$n \leq 50, m \leq 100, T \leq 20000$，时间限制 8s。

Codeforces 553E Solution

首先可以考虑一个比较暴力的 DP。设 $dp_{i,t}$ 表示在 t 时刻到达点 i 后，走完剩余路程的期望最小金钱花费。

Codeforces 553E Solution

首先可以考虑一个比较暴力的 DP。设 $dp_{i,t}$ 表示在 t 时刻到达点 i 后，走完剩余路程的期望最小金钱花费。

于是我们有这样的转移：

$$f_{u,t} = cost_i + \sum_{j=1}^{T} p_{i,j} \times f_{v,t+j}$$

Codeforces 553E Solution

首先可以考虑一个比较暴力的 DP。设 $dp_{i,t}$ 表示在 t 时刻到达点 i 后，走完剩余路程的期望最小金钱花费。

于是我们有这样的转移：

$$f_{u,t} = cost_i + \sum_{j=1}^{T} p_{i,j} \times f_{v,t+j}$$

其边界条件为 $f_{n,t} = 0\ (0 \le t \le T)$，$f_{u,t} = dis_u + x\ (T+1 \le t)$，其中 dis_u 表示从 u 到 n 经过边的最小条数。第二个边界的意思是说，如果现在走最短路仍然不能按时到达，不如走花费最小的边。

Codeforces 553E Solution

我们注意到这个转移中，$p_{i,j}$ 与 $f_{v,t+j}$ 下标第二维的差是相等的。因此我们可以将其中一个翻转一下，然后 FFT 优化。

Codeforces 553E Solution

我们注意到这个转移中，$p_{i,j}$ 与 $f_{v,t+j}$ 下标第二维的差是相等的。因此我们可以将其中一个翻转一下，然后 FFT 优化。

这里由于转移需要用到之前的答案，因此需要用分治 FFT。时间复杂度 $O(mT\log^2 T)$。

多项式求逆

利用 FFT，我们还可以实现多项式求逆。在讲解之前，先了解一下多项式逆元的定义。

多项式求逆

利用 FFT，我们还可以实现多项式求逆。在讲解之前，先了解一下多项式逆元的定义。

对于一个多项式 $A(x)$，如果存在 $B(x)$ 满足 B 的最高次数不大于 A 的最高次数，且

$$A(x)B(x) \equiv 1 (\mathrm{mod}\, x^n)$$

则称 $B(x)$ 为 $A(x)$ 在 $\mathrm{mod}\ x^n$ 意义下的逆元。

多项式求逆

首先对于 $n=1$ 的情况，当 $n=1$ 时，$A(x)\equiv c(\bmod x)$，所以此时 $B(x)=c^{-1}$。

多项式求逆

首先对于 $n=1$ 的情况，当 $n=1$ 时，$A(x)\equiv c(\bmod x)$，所以此时 $B(x)=c^{-1}$。

对于 $n>1$ 的情况，设 $B(x)$ 为 $A(x)$ 在 x^n 意义下的逆元，则由定义有：

$$A(x)B(x)\equiv 1(\bmod x^n)\equiv 1(\bmod x^{\lceil \frac{n}{2} \rceil})$$

多项式求逆

首先对于 $n=1$ 的情况，当 $n=1$ 时，$A(x)\equiv c(\bmod x)$，所以此时 $B(x)=c^{-1}$。

对于 $n>1$ 的情况，设 $B(x)$ 为 $A(x)$ 在 x^n 意义下的逆元，则由定义有：

$$A(x)B(x)\equiv 1(\bmod x^n)\equiv 1(\bmod x^{\lceil \frac{n}{2} \rceil})$$

设 $B'(x)$ 为 $A(x)$ 在 $x^{\lceil \frac{n}{2} \rceil}$ 意义下的逆元，则有：

$$A(x)B'(x)\equiv 1(\bmod x^{\lceil \frac{n}{2} \rceil})$$

两式相减，可得

$$B(x) - B'(x) \equiv 0 (\bmod x^{\lceil \frac{n}{2} \rceil})$$

两式相减，可得

$$B(x) - B'(x) \equiv 0 (\bmod x^{\lceil \frac{n}{2} \rceil})$$

两边平方，得

$$B^2(x) - 2B'(x)B(x) + B'^2(x) \equiv 0 (\bmod x^n)$$

两式相减，可得

$$B(x) - B'(x) \equiv 0 (\bmod x^{\lceil \frac{n}{2} \rceil})$$

两边平方，得

$$B^2(x) - 2B'(x)B(x) + B'^2(x) \equiv 0 (\bmod x^n)$$

两边同时乘上 $A(x)$，移项可得

$$B(x) \equiv 2B'(x) - A(x)B'^2(x) (\bmod x^n)$$

两式相减，可得

$$B(x) - B'(x) \equiv 0 (\bmod x^{\lceil \frac{n}{2} \rceil})$$

两边平方，得

$$B^2(x) - 2B'(x)B(x) + B'^2(x) \equiv 0 (\bmod x^n)$$

两边同时乘上 $A(x)$，移项可得

$$B(x) \equiv 2B'(x) - A(x)B'^2(x) (\bmod x^n)$$

这样我们就求出其逆元了，时间复杂度 $O(n \log n)$。

51Nod1514 美妙的序列

某个 $1 \to n$ 的排列如果满足：

在 $1 \to n-1$ 这些位置后面将序列断开，使得总可以从右边找到一个数，并且该数不比左边的所有的数都要大，则称该序列为“美妙的”。

51Nod1514 美妙的序列

某个 $1 \to n$ 的排列如果满足：

在 $1 \to n-1$ 这些位置后面将序列断开，使得总可以从右边找到一个数，并且该数不比左边的所有的数都要大，则称该序列为“美妙的”。

给出 n，求长度为 n 的“美妙的序列”的数量。

51Nod1514 美妙的序列

某个 $1 \to n$ 的排列如果满足：

在 $1 \to n-1$ 这些位置后面将序列断开，使得总可以从右边找到一个数，并且该数不比左边的所有的数都要大，则称该序列为“美妙的”。

给出 n，求长度为 n 的“美妙的序列”的数量。

多组数据，$n, T \leq 10^5$

51Nod1514 Solution

可以发现，满足题意等价与满足序列 1 到 i 的不为 1 到 i 的一个排列。

于是可以设 dp_i 表示长度为 i 时的答案。

51Nod1514 Solution

可以发现，满足题意等价与满足序列 1 到 i 的不为 1 到 i 的一个排列。于是可以设 dp_i 表示长度为 i 时的答案。

我们同样从反面计数转移，枚举不满足条件的排列中第一个不满足条件的位置，剩下的位置随便放，因此我们有转移方程：

$$dp_i = i! - \sum_{j=1}^{i-1} dp_j \times (i-j)!$$

51Nod1514 Solution

可以发现，满足题意等价与满足序列 1 到 i 的不为 1 到 i 的一个排列。于是可以设 dp_i 表示长度为 i 时的答案。

我们同样从反面计数转移，枚举不满足条件的排列中第一个不满足条件的位置，剩下的位置随便放，因此我们有转移方程：

$$dp_i = i! - \sum_{j=1}^{i-1} dp_j \times (i-j)!$$

这时我们当然可以选择分治 +FFT，时间复杂度 $O(n \log^2 n)$。

51Nod1514 Solution

如果要利用多项式求逆，该如何求解呢?

51Nod1514 Solution

如果要利用多项式求逆，该如何求解呢?

我们把转移式子移项，可得:

$$\sum_{j=1}^{i} dp_j \times (i-j)! = i!$$

51Nod1514 Solution

如果要利用多项式求逆，该如何求解呢?

我们把转移式子移项，可得:

$$\sum_{j=1}^{i} dp_j \times (i-j)! = i!$$

令 $F(x) = \sum_{i=1}^{n} dp_i \times x^i$，$G(x) = \sum_{i=0}^{n} i! \times x^i$，则有:

$$F(x) \times G(x) = G(x) - 1$$

51Nod1514 Solution

如果要利用多项式求逆，该如何求解呢?

我们把转移式子移项，可得:

$$\sum_{j=1}^{i} dp_j \times (i-j)! = i!$$

令 $F(x) = \sum_{i=1}^{n} dp_i \times x^i$，$G(x) = \sum_{i=0}^{n} i! \times x^i$，则有:

$$F(x) \times G(x) = G(x) - 1$$

所以有 $F(x) = 1 - \frac{1}{G(x)}$。

51Nod1514 Solution

如果要利用多项式求逆，该如何求解呢?

我们把转移式子移项，可得:

$$\sum_{j=1}^{i} dp_j \times (i-j)! = i!$$

令 $F(x) = \sum_{i=1}^{n} dp_i \times x^i$，$G(x) = \sum_{i=0}^{n} i! \times x^i$，则有:

$$F(x) \times G(x) = G(x) - 1$$

所以有 $F(x) = 1 - \frac{1}{G(x)}$。直接多项式求逆即可，复杂度为 $O(n \log n)$。

多项式求逆的小总结

其实许多分治 +FFT 的题目都可以利用多项式求逆来求解，并且复杂度会去掉一个 $\log$。

多项式求逆的小总结

其实许多分治 +FFT 的题目都可以利用多项式求逆来求解，并且复杂度会去掉一个 $\log$。

实现效率上，多项式求逆会占有一定的优势。本机测试分治 FFT 速度比多项式求逆大约慢 4 倍。

多项式求逆的小总结

其实许多分治 +FFT 的题目都可以利用多项式求逆来求解，并且复杂度会去掉一个 $\log$。

实现效率上，多项式求逆会占有一定的优势。本机测试分治 FFT 速度比多项式求逆大约慢 4 倍。

多项式求逆的一般套路也就是这样，设两个生成函数，并寻找其中的关系并求解。

多项式求逆的小总结

其实许多分治 +FFT 的题目都可以利用多项式求逆来求解，并且复杂度会去掉一个 $\log$。

实现效率上，多项式求逆会占有一定的优势。本机测试分治 FFT 速度比多项式求逆大约慢 4 倍。

多项式求逆的一般套路也就是这样，设两个生成函数，并寻找其中的关系并求解。

顺便说一句，刚刚的 BZOJ3456 也是可以多项式求逆的，感兴趣的可以自己去推一推。

FWT 引入

先来介绍一下，FWT 是做什么的。

FWT 引入

先来介绍一下，FWT 是做什么的。

FWT 的主要功能是求：$C_i = \sum_{j \otimes k} a_j \times b_k$，其中 $\otimes$ 是一种位运算符。

FWT 引入

先来介绍一下，FWT 是做什么的。

FWT 的主要功能是求：$C_i = \sum_{j \otimes k} a_j \times b_k$，其中 $\otimes$ 是一种位运算符。

注意到在普通 FFT 中，这个 $\otimes$ 运算符为 $+$。

FWT 引入

先来介绍一下，FWT 是做什么的。

FWT 的主要功能是求：$C_i = \sum_{j \otimes k} a_j \times b_k$，其中 $\otimes$ 是一种位运算符。

注意到在普通 FFT 中，这个 $\otimes$ 运算符为 $+$。

利用 FWT，同样可以在 $O(n \log n)$ 的时间内求出结果。

FWT 的具体实现

类似与我们做 FFT 的思路，我们考虑如何实现 FWT。

FWT 的具体实现

类似与我们做 FFT 的思路，我们考虑如何实现 FWT。

考虑一个变换：

$$DWT(A)_i = \sum_{j=0}^{n-1} A_j \times f_{i,j}$$

FWT 的具体实现

类似与我们做 FFT 的思路，我们考虑如何实现 FWT。

考虑一个变换：

$$DWT(A)_i = \sum_{j=0}^{n-1} A_j \times f_{i,j}$$

如果我们能找到这样一个变化系数 $f_{i,j}$，使得

$$DWT(A)_i \times DWT(B)_i = DWT(C)_i$$

那么我们就能用类似 FFT 的方法，先 DWT，再相乘，最后 IDWT。

FWT 的具体实现

类似与我们做 FFT 的思路，我们考虑如何实现 FWT。

考虑一个变换：

$$DWT(A)_i = \sum_{j=0}^{n-1} A_j \times f_{i,j}$$

如果我们能找到这样一个变化系数 $f_{i,j}$，使得

$$DWT(A)_i \times DWT(B)_i = DWT(C)_i$$

那么我们就能用类似 FFT 的方法，先 DWT，再相乘，最后 IDWT。

带入一下，上面那个条件其实就等价于：$f_{i,j} \times f_{i,k} = f_{i,j \otimes k}$。

FWT 的具体实现

在这里我直接给出这一构造，大家可以自己验证其正确性。

FWT 的具体实现

在这里我直接给出这一构造，大家可以自己验证其正确性。

- $\otimes$ 为与：$f_{i,j} = [i\&j = i]$
- $\otimes$ 为或：$f_{i,j} = [i\&j = j]$
- $\otimes$ 为异或：$f_{i,j} = (-1)^{\_\_\text{builtin\_popcount(i\&j)}}$

FWT 的具体实现

现在得到了这个 $f_{i,j}$，我们该怎么快速进行 DWT 呢?

FWT 的具体实现

现在得到了这个 $f_{i,j}$，我们该怎么快速进行 DWT 呢?

注意到这个 $f_{i,j}$ 有一个比较优秀的性质，就是将这个 i 和 j 的每一位二进制分解出来，变为 $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$，有

$$f_{i,j} = f_{i_1,j_1} \times f_{i_2,j_2} \times \cdots \times f_{i_k,j_k}$$

FWT 的具体实现

现在得到了这个 $f_{i,j}$，我们该怎么快速进行 DWT 呢?

注意到这个 $f_{i,j}$ 有一个比较优秀的性质，就是将这个 i 和 j 的每一位二进制分解出来，变为 $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$，有

$$f_{i,j} = f_{i_1,j_1} \times f_{i_2,j_2} \times \cdots \times f_{i_k,j_k}$$

利用这个性质，我们就可以开始推式子了。

FWT 的具体实现

设 $i_0 = [i \geq \frac{n}{2}]$，$i_1 = i \bmod \frac{n}{2}$，则有：

FWT 的具体实现

设 $i_0 = [i \geq \frac{n}{2}]$，$i_1 = i \bmod \frac{n}{2}$，则有：

$$
\begin{aligned}
DWT(A)_i &= \sum_{j=0}^{n-1} A_j \times f_{i,j} \\
&= \sum_{j=0}^{\frac{n}{2}-1} A_j \times f_{i,j} + \sum_{j=\frac{n}{2}}^{n-1} A_j \times f_{i,j} \\
&= \sum_{j=0}^{\frac{n}{2}-1} A_j \times f_{i_0,0} \times f_{i_1,j} + \sum_{j=\frac{n}{2}}^{n-1} A_j \times f_{i_0,1} \times f_{i_1,j-\frac{n}{2}} \\
&= f_{i_0,0} \times DWT(A_0)_{i_1} + f_{i_0,1} \times DWT(A_1)_{i_1}
\end{aligned}
$$

FWT 的具体实现

于是我们得到了 FWT 的这个递推式，然后按照 FFT 的模式去写就可以了。

FWT 的具体实现

于是我们得到了 FWT 的这个递推式，然后按照 FFT 的模式去写就可以了。

注意到这里的 A_0 和 A_1 与 FFT 中的有所不同，这个 A_0 与 A_1 并不需要二进制翻转。并且一般来讲，FWT 的题中，n 为 2 的整数次幂。因此代码会简洁不少。

FWT 的具体实现

于是我们得到了 FWT 的这个递推式，然后按照 FFT 的模式去写就可以了。

注意到这里的 A_0 和 A_1 与 FFT 中的有所不同，这个 A_0 与 A_1 并不需要二进制翻转。并且一般来讲，FWT 的题中，n 为 2 的整数次幂。因此代码会简洁不少。

至于 IDWT 的过程，限于篇幅不再赘述，方法与 FWT 是一样的。

FWT 的具体实现

于是我们得到了 FWT 的这个递推式，然后按照 FFT 的模式去写就可以了。

注意到这里的 A_0 和 A_1 与 FFT 中的有所不同，这个 A_0 与 A_1 并不需要二进制翻转。并且一般来讲，FWT 的题中，n 为 2 的整数次幂。因此代码会简洁不少。

至于 IDWT 的过程，限于篇幅不再赘述，方法与 FWT 是一样的。

~~实在不能理解背背代码也是挺好的~~

BZOJ4589 Hard Nim

Claris 和 NanoApe 在玩 Nim 游戏，共有 n 堆石头，Claris 先手。

BZOJ4589 Hard Nim

Claris 和 NanoApe 在玩 Nim 游戏，共有 n 堆石头，Claris 先手。

不同的初始局面，决定了最终的获胜者，有些局面下先拿的 Claris 会赢，其余的局面 Claris 会负。Claris 很好奇，如果这 n 堆石子满足每堆石子的初始数量是不超过 m 的质数，而且他们都会按照最优策略玩游戏，那么 NanoApe 能获胜的局面有多少种。

BZOJ4589 Hard Nim

Claris 和 NanoApe 在玩 Nim 游戏，共有 n 堆石头，Claris 先手。

不同的初始局面，决定了最终的获胜者，有些局面下先拿的 Claris 会赢，其余的局面 Claris 会负。Claris 很好奇，如果这 n 堆石子满足每堆石子的初始数量是不超过 m 的质数，而且他们都会按照最优策略玩游戏，那么 NanoApe 能获胜的局面有多少种。

由于答案可能很大，你只需要给出答案对 10^9+7 取模的值。

BZOJ4589 Hard Nim

Claris 和 NanoApe 在玩 Nim 游戏，共有 n 堆石头，Claris 先手。

不同的初始局面，决定了最终的获胜者，有些局面下先拿的 Claris 会赢，其余的局面 Claris 会负。Claris 很好奇，如果这 n 堆石子满足每堆石子的初始数量是不超过 m 的质数，而且他们都会按照最优策略玩游戏，那么 NanoApe 能获胜的局面有多少种。

由于答案可能很大，你只需要给出答案对 10^9+7 取模的值。

$n \leq 10^9, m \leq 50000$

BZOJ4589 Solution

算是一道模板题吧。

BZOJ4589 Solution

算是一道模板题吧。首先，只要所有石头的异或和为 0，那么这个状态就是 NanoApe 必胜的。所以我们只需要把一个数组中，所有质数位置上的值设为 1，然后和自己本身做 n 次 FWT，最后 0 位上的值就是最后的答案。

BZOJ4589 Solution

算是一道模板题吧。首先，只要所有石头的异或和为 0，那么这个状态就是 NanoApe 必胜的。所以我们只需要把一个数组中，所有质数位置上的值设为 1，然后和自己本身做 n 次 FWT，最后 0 位上的值就是最后的答案。

这里需要一个类似快速幂的东西。只需要先做一次 DWT，然后类似快速幂按位相乘，最后统一 IDWT 就可以了。

其他习题

由于本人比较菜，我找到的其他 FWT 的题目似乎都不会做 qwq

其他习题

由于本人比较菜，我找到的其他 FWT 的题目似乎都不会做 qwq

所以我还是把题目放在这里，欢迎大佬课后秒题。

其他习题

由于本人比较菜，我找到的其他 FWT 的题目似乎都不会做 qwq

所以我还是把题目放在这里，欢迎大佬课后秒题。感觉这里的题对于对 FWT 的理解要求还是比较高的。

其他习题

由于本人比较菜，我找到的其他 FWT 的题目似乎都不会做 qwq

所以我还是把题目放在这里，欢迎大佬课后秒题。感觉这里的题对于对 FWT 的理解要求还是比较高的。

- UOJ310 [UNR 2] 黎明前的巧克力
- UOJ348 [WC2018] 州区划分
- 51Nod1824 染色游戏

其他习题

由于本人比较菜，我找到的其他 FWT 的题目似乎都不会做 qwq

所以我还是把题目放在这里，欢迎大佬课后秒题。感觉这里的题对于对 FWT 的理解要求还是比较高的。

- UOJ310 [UNR 2] 黎明前的巧克力
- UOJ348 [WC2018] 州区划分
- 51Nod1824 染色游戏

做出来的大佬记得教教我啊 = =

Thanks