FFT 模式匹配

一.简介

FFT 即快速傅里叶变换，常用来加速多项式乘法，由此也可以变形用来解决其它问题。

二.多项式乘法

首先来看多项式乘法的定义：设有 2 个多项式

$$A = a_0 + a_1 * x + a_2 * x^2 + ... + a_n * x^n$$

$$B = b_0 + b_1 * x + b_2 * x^2 + ... + b_n * x^m$$

则 C = A * B 的第 p 项（从第 0 项开始）可以表示为：

$$C_p = \sum_{i=0}^{p} (a_i * b_{p-i}) x^p$$

，C 的项数为 n+m。

直接模拟多项式乘法的时间复杂度为 O(nm)，当 n 和 m>10000 时，肯定是不可取的。

这里可以用到 FFT 加速这一过程（原理就不说明了），可以使时间复杂度达到 O((n+m)log(n+m))

三.FFT 在字符串匹配上的应用

现在有 2 个字符串 S 和 T，令 m = |S|，n = |T|，设 n <= m

假设 S 的第 i 个位置的字符与 T 的第 j 个位置的字符匹配，则可以表示为 S[i] = T[j]

可知，假设 S[i ... i + n - 1]与 T[0 ... n - 1]匹配，则$_{j=0}$

上式是如何得到的呢？

因为需要将 S[i ... i + n - 1]与 T[0 ... n - 1]匹配，所以 S[i] = T[0],S[i+1] = T[1],...,S[i + n - 1] = T[n - 1]，得到$_{j=0}$

将上式变成平方和的形式可以去掉负数和正数相加恰好等于 0 的情况。

但是这个式子还不能满足使用 FFT 的条件！

由于 i + j + j 不是一个定值，因此我们无法将上式转化成多项式乘法的形式。但我们可以将 T 进行翻转后可以发现，只要 S[i] = T[n-1],S[i + 1] = T[n - 2],...,S[i + n - 1] = T[0]，就相当于翻转前的 T 与 S[i...i + n - 1]匹配成功了。

$$\sum_{j=0}^{n-1} (S[i+j] - T[n-j-1])^2 = 0$$

上式可变换为：

可以看到上式中 i + j + n - 1 - j = i + n - 1 是一个固定值，即求多项式乘法 F = S * T 后 F[i + n - 1]的值就是 S[i...i + n - 1]与 T[0...n - 1]匹配情况，只要 F[i + n - 1] = 0，则表示 S[i...i + n - 1]与 T[0...n - 1]完全匹配。

但是上式带有平方的形式，我们可以化成下面 3 个多项式和的形式分别求卷积即可：

$$\sum_{j=0}^{n-1} S[i+j]^2 + \sum_{j=0}^{n-1} T[n-j-1]^2 + \sum_{j=0}^{n-1} S[i+j] * T[n-j-1]$$

四.模板

```
const int MX = 1e6 + 5;
const int MAXL = MX * 4;
const double pi = acos (-1.0);
int len, res[MAXL], mx; //开大 4 倍
char s[MX], t[MX];
int a[MX], b[MX];
int T, n, m;
struct Complex {
    double r, i;
    Complex (double r = 0, double i = 0) : r (r), i (i) {};
    Complex operator+ (const Complex &rhs) {return Complex (r + rhs.r, i + rhs.i);}
    Complex operator- (const Complex &rhs) {return Complex (r - rhs.r, i - rhs.i);}
    Complex operator* (const Complex &rhs) {return Complex (r * rhs.r - i * rhs.i, i * rhs.r + r * rhs.i);}
} va[MAXL], vb[MAXL];
void rader (Complex F[], int len) { //len = 2^M,reverse F[i] with  F[j] j 为 i 二进制反转
    int j = len >> 1;
    for (int i = 1; i < len - 1; ++i) {
        if (i < j) swap (F[i], F[j]); // reverse
        int k = len >> 1;
        while (j >= k) {
            j -= k;
            k >>= 1;
        }

        if (j < k) j += k;
    }

}
void FFT (Complex F[], int len, int t) {

    rader (F, len);

    for (int h = 2; h <= len; h <<= 1) {
        Complex wn (cos (-t * 2 * pi / h), sin (-t * 2 * pi / h) );
        for (int j = 0; j < len; j += h) {
            Complex E (1, 0); //旋转因子
            for (int k = j; k < j + h / 2; ++k) {
```

```
                Complex u = F[k];
                Complex v = E * F[k + h / 2];
                F[k] = u + v;
                F[k + h / 2] = u - v;
                E = E * wn;
            }
        }
    }
    if (t == -1) //IDFT
        for (int i = 0; i < len; ++i)
            F[i].r /= len;
}
void Conv (Complex a[], Complex b[], int len) { //求卷积
    FFT (a, len, 1);
    FFT (b, len, 1);
    for (int i = 0; i < len; ++i) a[i] = a[i] * b[i];
    FFT (a, len, -1);
}
void gao() {
    Conv (va, vb, len);
    for (int i = 0; i < len; ++i) res[i] += va[i].r + 0.5;
}
void solve (int x, int y) {
    len = 1;
    mx = n + m;
    while (len <= mx) len <<= 1; //mx 为卷积后最大下标
    for (int i = 0; i < len; i++) va[i].r = va[i].i = vb[i].r = vb[i].i = 0;
    for (int i = 0; i < n; i++) va[i].r = a[i];
    for (int i = 0; i < m; i++) vb[i].r = b[i];
    gao();
}
```