AtCoder Typical Contest 001

C 高速フーリエ変換

AtCoder 株式会社

問題概要

AtCoder 食堂では,

- i 円の主菜が A_i 種類
- j 円の副菜が B_j 種類

ある.

ちょうど k 円になる, 主菜と副菜一つずつの組合せがいくつあるかを出力せよ.

畳込み

ちょうど k 円になる組合せの数を C_k とすると, 主菜で i 円の物を選んだ時, 副菜として $k-i$ 円の物を選べばよく,

$$C_k = \sum_{i=0}^{k} A_i B_{k-i}$$

となる. 但し, $A_0 = B_0 = 0$ とおく.

このような C を, A と B の畳込み (convolution) という.

畳込みから多項式乗算へ

ここで, A, B を係数とする多項式

$$g(x) = \sum_{i=0}^{N} A_i x^i, \qquad h(x) = \sum_{j=0}^{N} B_j x^j$$

を考えると, その積は

$$\begin{aligned}(g * h)(x) &= \sum_{k=0}^{2N} \left(\sum_{i=0}^{k} A_i B_{k-i} \right) x^k \\ &= \sum_{k=0}^{2N} C_k x^k\end{aligned}$$

となるから, これが計算出来れば, 答えがわかる.

多項式乗算

$g(x) * h(x)$ を高速に求めたい.

普通に書くと, こんな感じで $O(\deg(g) * \deg(h))$.

```
def multiply(g, h):
  f = [ 0 for _ in range(len(g) + len(h) - 1) ]
  for i in range(len(g)):
    for j in range(len(h)):
      f[i+j] += g[i] * h[j]
  return f
```

多項式の性質

$g(x) * h(x)$ は, $\deg(g) + \deg(h)$ 次の多項式.

よって, $\deg(g) + \deg(h) + 1$ 個の点 x_i での値 $f(x_i)$ が求まっていれば, これを通る h は一意.

例えば,

- 二点を決めると, それを同時に通る直線は一つ.
- 三点を決めると, それを同時に通る放物線は一つ.

高速多項式乗算の戦略

1. $n > \deg(g) + \deg(h)$ とし, n 個の点 $x_0, \dots, x_{n-1}$ を, 計算しやすいようにうまく選ぶ.
2. $g(x_0), \dots, g(x_{n-1})$ と, $h(x_0), \dots, h(x_{n-1})$ を計算する.
3. $(g * h)(x) = g(x) * h(x)$ を使って, $(g * h)(x_0), \dots, (g * h)(x_{n-1})$ を計算する.
4. うまいこと何かして, $(g * h)(x_0), \dots, (g * h)(x_{n-1})$ から $(g * h)(x)$ を復元する.

2 のように, 点での値を求めることを "評価" (evaluation), 4 のように, 点での値から元の多項式を復元することを "補完" (interpolation) と呼ぶ.

点の選び方

実際には, n が 2 の冪乗になるようにし, $x_0, \ldots, x_{n-1}$ としては 1 の n 乗根全体を選ぶ.

つまり, $\zeta_n = \exp(2\pi\sqrt{-1}/n)$ として, $x_i = \zeta_n^i$ とする.

ζ_n の性質

$\zeta_n = \exp(2\pi\sqrt{-1}/n)$ には, 次の性質がある.

- $\zeta_n^i = \zeta_n^j \Leftrightarrow i = j \bmod n.$
- "直交性" が成り立つ. すなわち,

$$\sum_{i=0}^{n-1} \left(\zeta_n^j\right)^i \left(\overline{\zeta_n^k}\right)^i = \sum_{i=0}^{n-1} \zeta_n^{i(j-k)}$$

$$= \begin{cases} n, & \text{if } j = k \bmod n, \\ 0, & \text{otherwise.} \end{cases}$$

(最後の $= 0$ は, 等比級数の和の公式から.)

ζ_n を ζ_n^{-1} で置き換えても, これらの性質は変わらない.

離散フーリエ変換

前述の通り, $x_i = \zeta_n^i$ として, 評価と補完をする. こうすると, 何がよいのかを見ていこう.

多項式 $f(x)$ に対し, $\hat{f}(t)$ を

$$\hat{f}(t) = \sum_{i=0}^{n-1} f(\zeta_n^i) t^i$$

で定める. つまり, 評価した各点での値を係数に持つ多項式である.

これを, f の離散フーリエ変換 (Discrete Fourier Transformation, DFT) と呼ぶ.

離散フーリエ変換

$f(x) = \sum_{j=0}^{n-1} c_j x^j$ とすると,

$$
\begin{aligned}
\widehat{f}(t) &= \sum_{i=0}^{n-1} f(\zeta_n^i) t^i \\
&= \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} c_j (\zeta_n^i)^j \right) t^i \\
&= \sum_{j=0}^{n-1} c_j \sum_{i=0}^{n-1} (\zeta_n^j t)^i
\end{aligned}
$$

離散フーリエ逆変換

$\widehat{f}(\zeta_n^{-k})$ を求めてみると,

$$\widehat{f}(\zeta_n^{-k}) = \sum_{j=0}^{n-1} c_j \sum_{i=0}^{n-1} (\zeta_n^j \zeta_n^{-k})^i$$

だが,

$$\sum_{i=0}^{n-1} \zeta_n^{i(j-k)} = \begin{cases} n, & \text{if } j = k \bmod n, \\ 0, & \text{otherwise} \end{cases}$$

だったから,

$$\widehat{f}(\zeta_n^{-k}) = nc_k.$$

離散フーリエ逆変換

よって, f の DFT

$$\hat{f}(t) = \sum_{i=0}^{n-1} f(\zeta_n^i) t^i$$

から,

$$f(x) = \frac{1}{n} \sum_{i=0}^{n-1} \hat{f}(\zeta_n^{-i}) x^i$$

と, ζ_n を ζ_n^{-1} で置き換えた DFT で $f(x)$ を復元出来る. これを, 離散フーリエ逆変換と呼ぶ.

積の離散フーリエ変換 (DFT)

さて, "多項式を評価した値" を係数としたのだから当然ではあるが, $\widehat{g * h}(t)$ は,

$$\widehat{g * h}(t) = \sum_{i=0}^{n-1} (g * h)(\zeta_n^i) t^i$$

$$= \sum_{i=0}^{n-1} g(\zeta_n^i) h(\zeta_n^i) t^i$$

と, $\widehat{g}$ と $\widehat{h}$ の係数毎の積で求められる.

離散フーリエ変換を使った乗算

結局, 多項式の積を求めるには,

1. $n > \deg(g) + \deg(h)$ となる 2 の冪乗を選ぶ.
2. 上の g, h に DFT をして $\widehat{g}(t), \widehat{h}(t)$ を計算する.
3. $\widehat{g}(t)$ と $\widehat{h}(t)$ を係数毎に掛け, $\widehat{g * h}(t)$ を求める.
4. $\widehat{g * h}(t)$ に inverse DFT をして $(g * h)(x)$ を復元する.

とすればよい.

離散フーリエ変換を使った乗算

擬似コードで書くと,

```
def multiply(g, h):
  n = pow_2_at_least(deg(g) + deg(h) + 1)
  # g, h は n-1 次になるように 0 を詰めておく.
  gg = dft(g, n)
  hh = dft(h, n)
  ff = [ gg[i] * hh[i] for i in range(n) ]
  return inverse_dft(ff, n)
```

高速フーリエ変換

あとは, DFT, inverse DFT を高速に求められればよい.

高速に DFT を求めるアルゴリズムを "高速フーリエ変換" (Fast Fourier Transformation, FFT) と呼ぶ.

inverse DFT は, DFT で出てくる ζ_n を全て ζ_n^{-1} で置き換え, 最後に n で割ればよいだけなので, 以下では DFT についてのみ解説する.

高速フーリエ変換

2 の冪乗 n と $n-1$ 次以下の多項式 $f(x)=\sum_{i=0}^{n-1} c_i x^i$ に対し,

$$f_0(x)=\sum_{i=0}^{n/2-1} c_{2i}x^i \ = c_0x^0+c_2x^1+c_4x^2+\dots,$$

$$f_1(x)=\sum_{i=0}^{n/2-1} c_{2i+1}x^i = c_1x^0+c_3x^1+c_5x^2+\dots$$

とすると,

$$f(x)=f_0(x^2)+xf_1(x^2)$$

で,f_0,f_1 はそれぞれ $n/2-1$ 次以下の多項式.

高速フーリエ変換

$\hat{f}$ を求めるには,

$$f(\zeta_n^0), f(\zeta_n^1), \ldots, f(\zeta_n^{n-1})$$

を求められればよかったが,$f(x) = f_0(x^2) + xf_1(x^2)$ だから,

$$f_0(\zeta_n^0), f_0(\zeta_n^2), \ldots, f_0(\zeta_n^{2(n-1)}),$$

$$f_1(\zeta_n^0), f_1(\zeta_n^2), \ldots, f_1(\zeta_n^{2(n-1)})$$

を求めればよい.

高速フーリエ変換

$\zeta_n^2 = \exp\left(2 * 2\pi\sqrt{-1}/n\right) = \exp\left(2\pi\sqrt{-1}/(n/2)\right) = \zeta_{n/2}$
だから,

$$f_0(\zeta_n^0), f_0(\zeta_n^2), \ldots, f_0(\zeta_n^{2(n-1)}),$$
$$f_1(\zeta_n^0), f_1(\zeta_n^2), \ldots, f_1(\zeta_n^{2(n-1)})$$

は,

$$f_0(\zeta_{n/2}^0), f_0(\zeta_{n/2}^1), \ldots, f_0(\zeta_{n/2}^{n-1}),$$
$$f_1(\zeta_{n/2}^0), f_1(\zeta_{n/2}^1), \ldots, f_1(\zeta_{n/2}^{n-1})$$

と同じ.

高速フーリエ変換

$\zeta_{n/2}$ は $n/2$ 乗すると 1 だから, $\zeta_{n/2}^{i+n/2} = \zeta_{n/2}^{i}$. よって,

$$f_0(\zeta_{n/2}^0), f_0(\zeta_{n/2}^1), \ldots, f_0(\zeta_{n/2}^{n-1}),$$
$$f_1(\zeta_{n/2}^0), f_1(\zeta_{n/2}^1), \ldots, f_1(\zeta_{n/2}^{n-1})$$

は, それぞれ前半と後半が同じで, 前半だけの

$$f_0(\zeta_{n/2}^0), f_0(\zeta_{n/2}^1), \ldots, f_0(\zeta_{n/2}^{n/2-1}),$$
$$f_1(\zeta_{n/2}^0), f_1(\zeta_{n/2}^1), \ldots, f_1(\zeta_{n/2}^{n/2-1})$$

を求めればよい.

高速フーリエ変換

よって, $n-1$ 次以下の多項式 f に対して

$$f(\zeta_n^0), f(\zeta_n^1), \dots, f(\zeta_n^{n-1})$$

を求めるには, 二つの $n/2-1$ 次以下の多項式 f_0, f_1 に対して

$$f_0(\zeta_{n/2}^0), f_0(\zeta_{n/2}^1), \dots, f_0(\zeta_{n/2}^{n/2-1}),$$
$$f_1(\zeta_{n/2}^0), f_1(\zeta_{n/2}^1), \dots, f_1(\zeta_{n/2}^{n/2-1})$$

を求めればよいことになった.

これは, サイズが半分になった同じ問題を二つ解けばよいということ!!

高速フーリエ変換

再帰的に行うと, 必要になる計算回数 $T(n)$ は,

$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2T(n/2) + O(n), & \text{otherwise} \end{cases}$$

で, これを解くと $T(n) = O(n \log n)$ になる.

高速フーリエ変換

以上のアルゴリズムを擬似コードで書くと,

```
def dft(f, n):
  if n == 1:
    return f
  f0 = [ f[2*i + 0] for i in range(n / 2) ]
  f1 = [ f[2*i + 1] for i in range(n / 2) ]
  f0 = dft(f0, n/2)
  f1 = dft(f1, n/2)
  zeta = complex(cos(2 * pi / n), sin(2 * pi / n))
  pow_zeta = 1
  for i in range(n)
    # この時点で, pow_zeta = pow(zeta, i)
    f[i] = f0[i % (n/2)] + pow_zeta * f1[i % (n/2)]
    pow_zeta *= zeta
  return f
```

発展的な話題

複素数以外の "環" での FFT

上の FFT は, 複素数だけでなく, 1 の原始 n 乗根(ちょうど n 乗すると 1 になるような要素)が存在する "可換環" で出来る.

これから, 例えば n で割ると 1 余る素数を法とする FFT が出来る事がわかる.

発展的な **FFT** アルゴリズム

今まで紹介した再帰的な FFT が基本だが, 他にも様々な FFT アルゴリズムがある.

- Cooley-Tukey FFT, Gentleman-Sande FFT
  - 再帰的 FFT を非再帰, in-place(入力の領域を使い回し, 余分な領域をあまり使わない)にしたもの. 競技プログラミングではよく用いられている.
- Stockham FFT
  - 上の二つと異なり, "ビット反転" が不要で, メモリアクセスがシーケンシャル.
  - その代わり, in-place でない.

発展的な **FFT** アルゴリズム

- 分割基底 FFT
  - f_0 と f_1 のようにニつに分割するのではなく, より多くの個数に分割する.
  - 4 つに分割する, 4-基底 FFT がよく用いられる.
- four-step FFT, six-step FFT
  - 約 $\sqrt{n}$ 個に分割し, 組み合わせる時にも FFT を用いる.
  - 並列化する時によいらしい.
- nine-step FFT
  - 約 $n^{1/3}$ ずつ, 三次元的に分割する.

参考文献

1. R. Crandall, C. Pomerance, 和田秀男 監訳,
   "素数全書: 計算からのアプローチ",
   朝倉書店, 2010, ISBN 978-4-254-11128-6.

2. R. Sedgewick,
   野下 浩平, 星 守, 佐藤 創, 田口 東 共訳,
   "アルゴリズムC <第3巻> グラフ・数理・トピックス",
   近代科学社, 1996, ISBN 978-4-764-90257-2