数論変換と形式的冪級数について

nPk

abstractkeywords

1 イントロダクション

1.1 組み合わせと多項式の関係

問題 1.1.1: 2つの数列 $\{a_i\}_{i=0}^{n-1}, \{b_i\}_{i=0}^{m-1}$ が与えられる. $\{c_k\}_{k=0}^{n+m-2} := \left\{\sum_{i+j=k} a_i b_j\right\}_{k=0}^{n+m-2}$ を求めよ.

問題 1.1.2: 2つの有限次元の多項式 $f(x) = \sum_{i=0}^{n-1} a_i x^i, g(x) = \sum_{i=0}^{m-1} b_i x^i$ が与えられる. これらの積 $h(x) = (f * g)(x) = \sum_{i=0}^{n+m-2} c_i x^i$ を求めよ.

問題 1.1.1 を愚直に解くと計算量は $\Theta(nm)$ です. 差分計算や分割統治などの簡単な高速化もないように思われます. 問題 1.1.2 はこれと完全に同じ問題ですが, 解析学の結果を利用できます.

定義 1.1.1 (離散フーリエ変換): $\mathbb{C}$ 上の $(n-1)$ 次多項式 f, $\mathbb{C}$ の原始 n 乗根を ω とする. このとき, f の離散フーリエ変換を次のように定める.

$$\mathcal{F}[f(x)] = \sum_{i=0}^{n-1} f(\omega^i) x^i \tag{1}$$

また, f の離散フーリエ逆変換を次のように定める.

$$\mathcal{F}^{-1}[f(x)] = \frac{1}{n} \sum_{i=0}^{n-1} f(\omega^{-i}) x^i \tag{2}$$

定理 1.1.1: f, g を $\mathbb{C}$ 上の $(n-1)$ 次多項式とする. 次が成り立つ.

1. $\mathcal{F}\left[\mathcal{F}^{-1}[f]\right] = \mathcal{F}^{-1}[\mathcal{F}[f]] = f$
2. $\mathcal{F}[f * g] = \mathcal{F}[f]\mathcal{F}[g]$

証明 *1.1.1*

1.
$$\begin{aligned}\mathcal{F}\left[\mathcal{F}^{-1}[f(x)]\right] &= \sum_{i=0}^{n-1}\left(\frac{1}{n}\sum_{j=0}^{n-1} f(\omega^{-j})\omega^{ij}\right)x^i \\ \therefore \mathcal{F}\left[\mathcal{F}^{-1}\left[f\left(\omega^{-k}\right)\right]\right] &= \frac{1}{n}\sum_{j=0}^{n-1} f(\omega^{-j})n\delta_{j,k} = f\left(\omega^{-k}\right)\ (0 \leq \forall k < n)\end{aligned} \tag{3}$$

$(n-1)$ 次多項式の異なる n 個の代表点が一致しているので, $\mathcal{F}\left[\mathcal{F}^{-1}[f]\right] = f$ が結論できる. $\mathcal{F}^{-1}[\mathcal{F}[f]] = f$ も同様にして証明できる. □

2.
$$\begin{aligned}\mathcal{F}[(f*g)(x)] &= \sum_{k=0}^{2n-2}(f*g)\left(\omega^k\right)x^k \\ &= \sum_{k=0}^{2n-2}\sum_{i+j=k} a_i b_j \omega^k x^k \\ &= \left(\sum_{i=0}^{n-1} a_i\omega^i x^i\right)\left(\sum_{j=0}^{n-1} b_j\omega^j x^j\right) \\ &= \mathcal{F}[f(x)]\mathcal{F}[g(x)]\end{aligned} \tag{4}$$

□

定理 1.1.1 より, 関数の積を求める方法が得られます.

系 1.1.1 (関数の積の計算): 次のようにして, 関数の積を計算できる.

1. ゼロ埋めにより f, g を $(n+m-2)$ 次の多項式に拡張し, 離散フーリエ変換を施す.
2. $\mathbb{C}$の原始 $(n+m-1)$ 乗根 w について, $\mathcal{F}\left[f(\omega^{-i})\right]\mathcal{F}\left[g(\omega^{-i})\right]\ (0 \leq i < n+m-1)$ を計算する.
3. 離散フーリエ逆変換により $(f*g)(x)$ を得る.

系 1.1.1 では係数の列から代表点の列を計算し, 項毎に積をとって新しい代表点の列を求め, さらに畳み込み後の係数列を求めています. 離散フーリエ変換・逆変換の定義から, 一方を求めるアルゴリズムがあれば, パラメーターを変えるだけで他方を求めるアルゴリズムも得られることが分かります.

1.2 参考資料

1. 一般の基数の数論変換アルゴリズムの比較
2. [多項式・形式的べき級数]（１）数え上げとの対応付け

2 離散フーリエ変換と数論変換

浮動小数の計算には誤差がつきものです. これを解決するために$\mathbb{C}$ではなく$\mathbb{Z}/p\mathbb{Z}$で計算することを考えます. これを数論変換と言います.

1. 原始根
2. 中国剰余定理による復元
3. NTT friendly prime number

3 Cooley-Tukey 型アルゴリズム

フーリエ変換・数論変換を計算する基本的なアルゴリズムです.

3.1 時間間引き

$(n-1)$ 次多項式 f の偶数次の項だけ集めたものをf_e, 奇数次の項だけを集めたものをf_oとかくと, $f(x) = f_e(x^2) + xf_o(x^2)$ が成り立ちます. 原始 n 乗根を w_n とします. n が偶数のとき, $\omega_n^2 = \omega_{\frac{n}{2}}$ であることに注意すると,

$$\begin{aligned} f\left(\omega_n^k\right) &= f_e\left(\omega_{\frac{n}{2}}^k\right) + \omega_n^k f_o\left(\omega_{\frac{n}{2}}^k\right) \\ f\left(\omega_n^{k+\frac{n}{2}}\right) &= f_e\left(\omega_{\frac{n}{2}}^k\right) - \omega_n^k f_o\left(\omega_{\frac{n}{2}}^k\right) \end{aligned} \tag{5}$$

が成り立ちます. ここで, $\omega_n^{\frac{n}{2}} = -1, \omega_{\frac{n}{2}}^{\frac{n}{2}} = 1$ の関係を使いました. (5) は分轄統治で数論変換を実行できることを意味しています. 計算量の漸化式は,

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \tag{6}$$

なので, $T(n) = \Theta(n \log n)$ です. 差分計算ですべての ω_n^k を $\Theta(n)$ で計算できることに注意してください. これを時間間引きアルゴリズムと言います.

以下では n が 2 冪であると仮定します. ゼロ埋めによって, いつでも 2 冪にできます.

定理 3.1.1: 関数の積を $\Theta(n \log n)$ で計算できる.

証明 *3.1.1* 系 1.1.1 より明らか.

3.2 ビット反転順序

メモリを節約するために時間間引きをその場で計算したいです. サイズ $\frac{n}{2}$ の数論変換の k 番目の代表点から, サイズ n の数論変換の k 番目と $k + \frac{n}{2}$ 番目の代表点が得られることから, 配列の前半に偶数次の結果を格納し, 後半にの結果を奇数次の結果を格納すればよいことが分かります. 形式的には `(i & 1, i >> 1)` でソートされていればよいです. 再帰的に計算することを考

えると, はじめに `i.reverse()` でソートしてしまえばよいです. これをビット反転順序といいます. 以上より, 追加の作業メモリが不要になりました.

アルゴリズム 3.2.1 (時間間引きアルゴリズム):

```rust
fn ntt_t<T>(values: &mut [T] /* in bit reversed order */) {
  assert!(values.len().is_power_of_two());

  let mut width = 1;
  while (width << 1) <= values.len() {
    let dw = /* 原始 width 乗根 */;
    for pair in values.chunks_exact_mut(width << 1) {
      let (prefix, suffix) = pair.split_at_mut(width);
      let mut w = 1;
      for i in 0..width {
        (prefix[i], suffix[i]) = (
          prefix[i] + w * suffix[i],
          prefix[i] - w * suffix[i],
        );
        w += dw;
      }
    }
    width <<= 1;
  }
}
```

3.3 最適化

3.3.1 掛け算の削減

ω_n^k を回転因子と言います. アルゴリズム 3.2.1 で回転因子は高々 n 種類しか登場しませんが, コードの 15 行目で $n \log n$ 回の計算をしています. そこで, コードの 2 番目と 3 番目のループを入れ替えます. 1 番目のループの各イテレーションで `width` 回ずつ回転因子を計算すればよいので, 回転因子の計算回数は $2n - 1$ 回になります.

回転因子の計算回数を減らすことに成功しましたが, メモリへのアクセスがシーケンシャルではなくなってしまいました. これを解決するために, 各 `pair` の `prefix` と `suffix` の i 番目をまとめてしまいたいです. 再帰のベースケース

を考えます. 登場する回転因子は $\omega_2^0 = 1$ だけなので, 正順の配列を前半と後半で分ければシーケンシャルアクセスが実現できます. 一段上の再帰過程では, 配列を 4 等分して前から順にペアをとります. このようなペアの取り方は アルゴリズム 3.2.1 でのペアの取り方の逆になっています. ビット反転順序のビット反転順序は正順なので, 改良版のアルゴリズムは正順の入力からビット反転順序の出力を与えます.

アルゴリズム 3.3.1.1 (時間間引きアルゴリズム・修正版): 正順の入力から, ビット反転順序の出力を得る.

```rust
fn ntt_t<T>(values: &mut [T]) {
  assert!(values.len().is_power_of_two());

  let mut width = value.len() >> 1;
  while width > 0 {
    for pair in values.chunks_exact_mut(width << 1) {
      let w = /* 適切な回転因子 */;
      let (prefix, suffix) = pair.split_at_mut(width);
      for i in 0..width {
        (prefix[i], suffix[i]) = (
          prefix[i] + w * suffix[i],
          prefix[i] - w * suffix[i],
        );
      }
    }
    width >>= 1
  }
}
```

回転因子の定義から, ビット反転順序の数論変換を得る際には回転因子もビット反転順序で登場します.

3.3.2 回転因子の差分計算

アルゴリズム 3.3.1.1 では回転因子がビット反転順序で登場します. メモ化するためには $\Theta(n)$ の作業メモリが必要なので, 差分計算したいです. アルゴリズム 3.3.1.1 の 2 番目のループを考えます. ループの開始時点で回転因子はいつも $w_n^0 = 1$ です. 1 番目のループのインデックスを i, 2 番目のループのインデックスを j とおきます. 回転因子の漸化式は $\Omega_{i,j+1} =$

$\Omega_{i,j}\omega_{n2^{-i}}^{\mathrm{reverse}(i,j+1)-\mathrm{reverse}(i,j)}$ です. ここで, $\mathrm{reverse}(i,j)$ は $(i-1)$ ビット整数としての j のビット反転です. たとえば, $j=****1011_{(2)}$ とすると $j+1=****1100_{(2)}$ なので,

$$\begin{aligned}\mathrm{reverse}(i,j+1)-\mathrm{reverse}(i,j) &= 0011****_{(2)} - 1101****_{(2)} \\ &= 2^{i-4}-2^{i-3}-2^{i-2} \\ \therefore \Omega_{i,j+1} &= \Omega_{i,j}\omega_4^{-1}\omega_8^{-1}\omega_{16}^{1}\end{aligned} \tag{7}$$

とかけます. この係数は明らかに `i.trailing_ones()` で決まるので, $\Theta(\log n)$ の作業メモリで差分計算ができます. ワードサイズ程度の大きさの数値で十分な場合, 現実的な時間でコンパイル時計算をすることができます.

アルゴリズム 3.3.2.1 (時間間引きアルゴリズム・修正版 2): 正順の入力から, ビット反転順序の出力を得る.

```rust
fn ntt_t<T>(values: &mut [T]) {
  assert!(values.len().is_power_of_two());

  let mut width = value.len() >> 1;
  while width > 0 {
    let mut w = 1;
    for (i, pair) in values.chunks_exact_mut(width << 1).enumerate() {
      let (prefix, suffix) = pair.split_at_mut(width);
      for i in 0..width {
        (prefix[i], suffix[i]) = (
          prefix[i] + w * suffix[i],
          prefix[i] - w * suffix[i],
        );
      }
      // `RATE`はコンパイル時に計算しておく
      w *= RATE[i.trailing_ones() as usize];
    }
    width >>= 1
  }
}
```

3.3.3 並べ替えの削除

アルゴリズム 3.3.2.1 より正順の係数列から, 代表点の列をビット反転順序で得ることができます. アルゴリズム 3.2.1 で数論逆変換を行うと, ビット反転順序の代表点の列から正順の係数列を得ることができます. 以上より, ビット反転順序に並べ替えることなしに関数の積を計算できます. しかしながら, アルゴリズム 3.2.1 は最適化されていません.

3.4 周波数間引き

ビット反転順序の入力をうけとり, 正順の出力を返す数論変換アルゴリズムがあれば, 最適化を進めることができます. 時間引きアルゴリズムは $f(k)$ と $f(k+\frac{n}{2})$ を部分問題から求めます. 部分問題から $f(2k)$ と $f(2k+1)$ を求めるアルゴリズムがあれば, それが所望のアルゴリズムです. $f(x)=\sum_{i=0}^{\frac{n}{2}-1}\left(a_i+a_{i+\frac{n}{2}}x^{\frac{n}{2}}\right)x^i$ より,

$$\begin{aligned} f\left(\omega_n^{2k}\right) &= \sum_{i=0}^{\frac{n}{2}-1}\left(a_i+a_{i+\frac{n}{2}}\right)\omega_{\frac{n}{2}}^{ik} \\ f\left(\omega_n^{2k+1}\right) &= \sum_{i=0}^{\frac{n}{2}-1}\left(\left(a_i-a_{i+\frac{n}{2}}\right)\omega_n^i\right)\omega_{\frac{n}{2}}^{ik} \end{aligned} \tag{8}$$

を得ます. これより, 新たな分割統治アルゴリズムが得られます. 計算量は $\Theta(n\log n)$ です. これを周波数間引きアルゴリズムといいます. 時間間引きアルゴリズムでは部分問題の出力から元の数論変換を計算していました. 一方で, 周波数間引きアルゴリズムでは部分問題の入力を計算していくと自動的に数論変換が完了しています.

周波数間引きアルゴリズムをその場で実行するには, 部分問題の解が交互に並んでいればよいです. a_i と $a_{i+\frac{n}{2}}$ が隣接しているので, ビット反転順序の入力から正順の出力が得られることが分かります. また, 各部分問題に対応する配列の i 番目がまとまっているので, 時間間引きアルゴリズムで検討した回転因子の計算回数やメモリアクセスの最適化がされています.

アルゴリズム 3.4.1 (周波数間引きアルゴリズム): ビット反転順序の入力から正順の出力を得る.

```rust
fn ntt_f<T>(values: &mut [T] /* in bit-reversed order*/) {
  assert!(values.len().is_power_of_two());

  let mut width = 1;
```

```
5     while (width << 1) <= values.len() {
6       let mut w = 1;
7       for (i, pair) in values.chunks_exact_mut(width <<
        1).enumerate() {
8         let (prefix, suffix) = pair.split_at_mut(width);
9         for i in 0..width {
10          (prefix[i], suffix[i]) = (
11            prefix[i] + suffix[i],
12            (prefix[i] - suffix[i]) * w,
13          );
14        }
15        // `RATE`はコンパイル時に計算しておく
16        w *= RATE[i.trailing_ones() as usize];
17      }
18      width <<= 1
19    }
20  }
```

3.5 追加の最適化

アルゴリズム 3.3.2.1 と アルゴリズム 3.4.1 を組み合わせることで, 関数の積を求めるアルゴリズムを最適化できました. 追加の最適化手法をいくつか述べます.

3.5.1 掛け算の削除

2 番目のループの回転因子は 1 から始まります. この掛け算は容易に削除でき, 条件分岐も必要ありません. 掛け算を $3(n-1)$ 回分削除できました.

3.5.2 SIMD 命令の活用

3 番目のループでは SIMD 命令を活用できます. SIMD 命令に除算はないので, モンゴメリ剰余乗算などを活用して計算する必要があります. 法はコンパイル時に指定できるので, 必要なパラメーターを関連定数として与えておくとよいです. 邪道ですが, 最適化された命令セットからパラメーターを拝借することもできます.

3.6 まとめ

n が 2 冪のときの数論変換を $\Theta(n \log n)$ で計算するアルゴリズムを得ました. 掛け算やメモリアクセスの回数が少なくなるように最適化しました. また, SIMD 命令を活用してさらに高速化することが分かりました.

1. べき乗をとると MLE するかも？

3.7 参考資料

1. 爆速な NTT を実装したい
2. FFT (高速フーリエ・コサイン・サイン変換) の概略と設計法
3. 一般の基数の数論変換アルゴリズムの比較

4 Prime Factor 型アルゴリズム

Cooley-Tukey 型 NTT では 2 冪サイズになるまでゼロ埋めする必要がありました. このとき, 配列のサイズは平均で 1.5 倍になります. 4 章では, Prime Factor 型アルゴリズムを紹介します. これと Cooley-Tukey 型アルゴリズムを組み合わせることで, ゼロ埋めする要素数を高々 $\sqrt{n}$ 個に抑えることができることを示します.

4.1 アルゴリズム

4.2 ゼロ埋めの最適化

4.3 Sparse NTT

最後の FFT はゼロが多い

5 形式的冪級数