Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Численные методы»
Тема: «Применение БПФ для перемножения длинных чисел»

Студент: М. А. Инютин
Преподаватель: Д. Л. Ревизников
Группа: М8О-307Б-19

Дата:
Оценка:
Подпись:

Москва, 2022

Постановка задачи

Задача: Даны два целых числа a и b. Требуется найти их произведение $a \cdot b$.

Сравнить производительноть простоого перемножения в столбик и с использованием быстрого преобразования Фурье.

1 Описание

Наивное перемножение

Обозначим n и m — количество разрядов в числе a и b соответственно. Для перемножения двух чисел в столбик требуется сопоставлять каждому разряду числа a каждый разряд числа b. То есть временная сложность такого алгоритма $O(n \cdot m)$.

Дискретное преобразование Фурье (ДПФ)

Пусть $A(x)$ — многочлен $n-1$ степени:

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

Будем считать, что n является степенью двойки. Если это не так, то мы можем дополнить исходный многочлен недостающими членами с нулевыми коэффициентами.

Обозначим $\omega_{n,k}$ — k-й комплексный корень степени n из единицы. Так как $1 = e^{2\pi i}$, то решая уравнение $z^n = 1 = e^{2\pi i}$ получим $z_k = e^{k \cdot \frac{2\pi i}{n}} = \omega_{n,k}$, где $k \in \{0, 1, 2, \dots n-1\}$. Обозначим $\omega_n = \omega_{n,1}$. По свойству степени $\omega_{n,k} = \omega_n^k$.

Назовём дискретным преобразованием Фурье многочлена $A(x)$ вектор из n чисел:

$$DFT(a_0, a_1, a_2, \dots a_{n-1}) = (A(\omega_n^0), A(\omega_n^1), A(\omega_n^2), \dots A(\omega_n^{n-1}))^T = (y_0, y_1, y_2, \dots y_{n-1})^T$$

Заметим, что для ω_n^k значение многочлена равно:

$$\begin{aligned} A(\omega_n^k) &= a_0 \cdot (\omega_n^k)^0 + a_1 \cdot (\omega_n^k)^1 + a_2 \cdot (\omega_n^k)^2 + a_3 \cdot (\omega_n^k)^3 + \cdots + a_{n-1} \cdot (\omega_n^k)^{n-1} = \\ &= a_0 \cdot \omega_n^0 + a_1 \cdot \omega_n^k + a_2 \cdot \omega_n^{2 \cdot k} + a_3 \cdot \omega_n^{3 \cdot k} + \cdots + a_{n-1} \cdot \omega_n^{(n-1) \cdot k} \end{aligned}$$

Тогда преобразование можно записать в матричном виде:

$$DFT(a_0, a_1, a_2, \dots a_{n-1}) = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{pmatrix} = W \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{pmatrix}$$

Здесь W — матрица Вандермонда:

$$W = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{(n-1)} \\ \omega_n^0 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2\cdot(n-1)} \\ \omega_n^0 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \dots & \omega_n^{3\cdot(n-1)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \omega_n^0 & \omega_n^{(n-1)} & \omega_n^{2\cdot(n-1)} & \omega_n^{3\cdot(n-1)} & \dots & \omega_n^{(n-1)^2} \end{pmatrix}$$

Умножим обе части слева на W^{-1}, получим:

$$W^{-1} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{pmatrix}$$

W^{-1} имеет следующий вид:

$$W^{-1} = \frac{1}{n} \cdot \begin{pmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^0 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \dots & \omega_n^{-(n-1)} \\ \omega_n^0 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \dots & \omega_n^{-2\cdot(n-1)} \\ \omega_n^0 & \omega_n^{-3} & \omega_n^{-6} & \omega_n^{-9} & \dots & \omega_n^{-3\cdot(n-1)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \omega_n^0 & \omega_n^{-(n-1)} & \omega_n^{-2\cdot(n-1)} & \omega_n^{-3\cdot(n-1)} & \dots & \omega_n^{-(n-1)^2} \end{pmatrix}$$

Определим обратное преобразование Фурье для вектора из n чисел:

$$InverseDFT(y_0, y_1, y_2, \dots y_{n-1}) = (a_0, a_1, a_2, \dots a_{n-1})^T$$

Или в матричном виде:

$$InverseDFT(y_0, y_1, y_2, \dots y_{n-1}) = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{pmatrix} = W^{-1} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{pmatrix}$$

Очевидно, что $InverseDFT(DFT(A)) = A$.

Вычисление $DFT(A)$ простым перемножение матрицы на вектор требует $O(n^2)$ операций, однако можно добиться существенного ускорения.

Быстрое преобразование Фурье (БПФ)

Разобьём исходный многочлен $A(x)$ на два многочлена $A_0(x)$ и $A_1(x)$, содержащие чётные и нечётные коэффициенты соответственно:

$$A_0(x) = a_0 + a_2 \cdot x + a_4 \cdot x^2 + \cdots + a_{n-2} \cdot x^{\frac{n}{2}-1}$$

$$A_1(x) = a_1 + a_3 \cdot x + a_5 \cdot x^2 + \cdots + a_{n-1} \cdot x^{\frac{n}{2}-1}$$

Тогда мы можем вычислить значение исходного многочлена следующим образом:

$$A(x) = A_0(x^2) + x \cdot A_1(x^2)$$

Вычислим дискретное преобразование Фурье для многочленов $A_0(x)$ и $A_1(x)$:

$$DFT(A_0) = (y_0^0, y_1^0, y_2^0, \dots y_{\frac{n}{2}-1}^0)^T$$

$$DFT(A_1) = (y_0^1, y_1^1, y_2^1, \dots y_{\frac{n}{2}-1}^1)^T$$

Здесь y^0 и y^1 — значения ДПФ для $A_0(x)$ и $A_1(x)$, а не показатели степени при y.
Теперь вычислим $DFT(A)$, используя $DFT(A_0)$ и $DFT(A_1)$, используя следующие три важных свойства:

$$\omega_{n,k}^2 = (e^{k \cdot \frac{2\pi i}{n}})^2 = e^{2 \cdot k \cdot \frac{2\pi i}{n}} = e^{k \cdot \frac{2\pi i}{\frac{n}{2}}} = \omega_{\frac{n}{2},k}$$

$$\omega_{n,k+\frac{n}{2}} = e^{(k+\frac{n}{2}) \cdot \frac{2\pi i}{n}} = e^{k \cdot \frac{2\pi i}{n}} \cdot e^{\frac{n}{2} \cdot \frac{2\pi i}{n}} = e^{k \cdot \frac{2\pi i}{n}} \cdot e^{\pi i} = e^{k \cdot \frac{2\pi i}{n}} \cdot (-1) = -e^{k \cdot \frac{2\pi i}{n}} = -\omega_{n,k}$$

$$\omega_{n,k+\frac{n}{2}}^2 = (-\omega_{n,k})^2 = \omega_{n,k}^2 = \omega_{\frac{n}{2},k}$$

Пусть $k \in \{0, 1, 2, \dots \frac{n}{2} - 1\}$, тогда, используя первое свойство, получим:

$$y_k = A(\omega_{n,k}) = A_0(\omega_{n,k}^2) + \omega_{n,k} \cdot A_1(\omega_{n,k}^2) = A_0(\omega_{\frac{n}{2},k}) + \omega_{n,k} \cdot A_1(\omega_{\frac{n}{2},k})$$

Степени многочленов $A_0(x)$ и $A_1(x)$ равны $\frac{n}{2}-1$, значит $y_k^0 = A_0(\omega_{\frac{n}{2},k})$ и $y_k^1 = A_1(\omega_{\frac{n}{2},k})$.
Подставим в выражение выше, получим:

$$y_k = y_k^0 + \omega_{n,k} \cdot y_k^1$$

Таким образом, мы вычислили половину значений $DFT(A)$, используя значения $DFT(A_0)$ и $DFT(A_1)$.
Используем второе и третье свойства:

$$y_{k+\frac{n}{2}} = A(\omega_{n,k+\frac{n}{2}}) = A_0(\omega_{n,k+\frac{n}{2}}^2) + \omega_{n,k+\frac{n}{2}} \cdot A_1(\omega_{n,k+\frac{n}{2}}^2) = A_0(\omega_{\frac{n}{2},k}) - \omega_{n,k} \cdot A_1(\omega_{\frac{n}{2},k})$$

Подставим y_k^0 и y_k^1 в выражение:

$$y_{k+\frac{n}{2}} = y_k^0 - \omega_{n,k} \cdot y_k^1$$

Итак, мы получили $DFT(A)$ с помощью значений $DFT(A_0)$ и $DFT(A_1)$. То есть мы разбили исходную задачу на две поздачи вдвое меньшего размера и, решив их, получили решение исходной задачи. Такая схема называется «разделяй-и-властвуй» и имеет известную вычислительную сложность $O(n \cdot \log(n))$.

Всё вышеописанное называется быстрым преобразование Фурье.

Обратное быстрое преобразование Фурье

Выше дано определение ДПФ в матричном виде. БПФ является методом быстрого вычисления ДПФ. Обратное преобразование Фурье строится так же перемножением матрицы на вектор. Для быстрого вычисления обратного преобразования Фурье достаточно обозначить $\omega_{n,k} = \omega_n^{-k} = e^{-k \cdot \frac{2\pi i}{n}}$ в формулах для БПФ, получаться аналогичные формулы. На последнем шаге вычисления обратного преобразования Фурье нужно не забыть разделить все полученные коэффициенты на n.

Листинг рекурсивного БПФ

Ниже приведена рекурсивная и простая реализация БПФ. Функция принимает вектор комплексных чисел и записывает в него же результат вычисления прямого или обратного БПФ.

```
using complex = std::complex<double>;
using vc = std::vector<complex>;

const double PI = std::acos(-1);

void fft(vc & a, bool invert) {
    size_t n = a.size();
    if (n == 1) {
        return;
    }
    vc a0(n / 2), a1(n / 2);
    for (size_t i = 0, j = 0; j < n; ++i, j += 2) {
        a0[i] = a[j];
        a1[i] = a[j + 1];
    }
    fft(a0, invert);
    fft(a1, invert);
    double phi = 2.0 * PI / n;
    if (invert) {
        phi = -phi;
    }
    complex w(1), wn(std::cos(phi), std::sin(phi));
    for (size_t i = 0; i < n / 2; ++i) {
        a[i] = a0[i] + w * a1[i];
        a[n / 2 + i] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2.0;
            a[n / 2 + i] /= 2.0;
        }
        w *= wn;
    }
}
```

Ускорение алгоритма

Заметим, что в приведённой реализации используется $O(n \cdot \log(n))$ памяти, из-за чего константа в асимптотике алгоритма довольно велика. Можно отказаться от рекурсивного выделения памяти, если переупорядочить элементы в исходном векторе.

Рассмотрим самый нижний уровень рекурсии и то, какие коэффициенты исходного многочлена будут использоваться при вычислении.

Изначальный вектор (окружен фигурными скобками)

$$\{a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$$

распадается на два вызова (окружены квадратными скобками)

$$\{[a_0, a_2, a_4, a_6], [a_1, a_3, a_5, a_7]\}$$

На следующем уровне рекурсии

$$\{[a_0, a_2, a_4, a_6], [a_1, a_3, a_5, a_7]\}$$

распадается на четыре вызова (окружены круглыми скобками)

$$\{[(a_0, a_4), (a_2, a_6)], [(a_1, a_5), (a_3, a_7)]\}$$

Полученный порядок называется поразрядно обратной перестановкой. Если мы посмотрим на какой позиции стоит тот или иной элемент, окажется, что если мы выполним реверс бит в двоичной записи изначальных позиций, то получим позиции на последнем уровне рекурсии. Например, $6_{10} = 110_2$, выполним реверс бит, получим $011_2 = 3_{10}$, что соответствует индексы элемента a_6 на последнем уровне рекурсии.

Упорядочим элементы вектора a как на последнем слою. Вычислим значения ДПФ для пар, окруженных круглыми скобками и запишем их вместо самих элементов, получим:

$$\{[(y_0^0, y_1^0), (y_0^1, y_0^1)], [(a_1, a_5), (a_3, a_7)]\}$$

Теперь запишем новые значение ДПФ на позиции самих элементов:

$$\{[(y_0^0 + \omega_{4,0} \cdot y_0^1, y_1^0 + \omega_{4,1} \cdot y_1^1), (y_0^0 - \omega_{4,0} \cdot y_0^1, y_1^0 - \omega_{4,1} \cdot y_1^1)], [(a_1, a_5), (a_3, a_7)]\}$$

Проделаем ту же операцию справа, получим:

$$\{[y_0^0, y_1^0, y_2^0, y_3^0], [y_0^1, y_1^1, y_2^1, y_3^1]\}$$

Видно, что элементы снова стоят подходящим образом, можно записывать значения ДПФ на позиции самих элементов:

$$\{[y_0^0 + \omega_{8,0} \cdot y_0^1, y_1^0, y_2^0, y_3^0], [y_0^0 - \omega_{8,0} \cdot y_0^1, y_1^1, y_2^1, y_3^1]\}$$

После всех вычислений получим:

$$\{y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7\}$$

Таким образом мы смогли вычислить значения ДПФ без использования дополнительной памяти. Такой подход уменьшает константу в асимптотике. Временная сложность по-прежнему $O(n \cdot \log(n))$, а вот пространственная $O(1)$.

Листинг нерекурсивного БПФ

Ниже приведена нерекурсивная реализация БПФ.

```
using complex = std::complex<double>;
using vc = std::vector<complex>;

const double PI = std::acos(-1);

int rev_bits(int x, int lg_n) {
    int y = 0;
    for (int i = 0; i < lg_n; ++i) {
        y = y << 1;
        y ^= (x & 1);
        x = x >> 1;
    }
    return y;
}

void fft(vc & a, bool invert) {
    int n = a.size();
    int lg_n = 0;
    while ((1 << lg_n) < n) {
        ++lg_n;
    }
    for (int i = 0; i < n; ++i) {
        if (i < rev_bits(i, lg_n)) {
            swap(a[i], a[rev_bits(i, lg_n)]);
        }
    }
    for (int layer = 1; layer <= lg_n; ++layer) {
        int cluster = 1 << layer;
        double phi = (2.0 * PI) / cluster;
        if (invert) {
            phi *= -1;
        }
        complex wn = complex(std::cos(phi), std::sin(phi));
        for (int i = 0; i < n; i += cluster) {
            complex w(1);
            for (int j = 0; j < cluster / 2; ++j) {
                complex u = a[i + j];
```

```
38                complex v = a[i + j + cluster / 2] * w;
39                a[i + j] = u + v;
40                a[i + j + cluster / 2] = u - v;
41                w *= wn;
42            }
43        }
44    }
45    if (invert) {
46        for (int i = 0; i < n; ++i) {
47            a[i] /= n;
48        }
49    }
50 }
```

Перемножение двух чисел с использованием БПФ

Пусть $a = 7310$ и $b = 2468$. Можно записать $a = 7 \cdot 10^3 + 3 \cdot 10^2 + 1 \cdot 10 + 0$ и $b = 2 \cdot 10^3 + 4 \cdot 10^2 + 6 \cdot 10 + 8$.

Пусть $A(x) = 0 + 1 \cdot x + 3 \cdot x^2 + 7 \cdot x^3$ и $B(x) = 8 + 6 \cdot x + 4 \cdot x^2 + 2 \cdot x^3$. Тогда $A(10) = a$ и $B(10) = b$. Если мы вычислим ДПФ для многочленов $A(x)$ и $B(x)$, то перемножив их значения ДПФ в одних и тех же точках, мы получим значения ДПФ для $(A \cdot B)(x)$, то есть

$$DFT(A \cdot B) = DFT(A) \cdot DFT(B)$$

Применив обратное преобразование Фурье к полученному и вектору, мы получим коэффициенты многочлена $(A \cdot B)(x)$. Так мы смогли перемножить два многочлена, используя БПФ:

$$(A \cdot B)(x) = InverseDFT(DFT(A \cdot B)) = InverseDFT(DFT(A) \cdot DFT(B))$$

Так как A и B изначально представляли числа a и b, выполним перенос разрядов, чтобы получить многочлен, предсталвяющий $a \cdot b$.

Такой подход позволяет перемножать два многочлена за $O(n \cdot \log(n))$. Мы смогли представить два числа в виде многочленов, таким образом смогли перемножить два числа за такую же асимптотику $O(n \cdot \log(n))$.

2 Исходный код

Ниже приведена реализация перемножения чисел в столбик.

```
std::string slow_mult(const std::string & a, const std::string & b) {
    size_t n = 4 * std::max(a.size(), b.size());
    vi res(n);
    for (size_t i = 0; i < a.size(); ++i) {
        int64_t ai = a[i] - '0';
        for (size_t j = 0; j < b.size(); ++j) {
            int64_t bj = b[j] - '0';
            res[(b.size() - 1 - j) + (a.size() - 1 - i)] += ai * bj;
        }
    }
    for (size_t i = 0; i < n - 1; ++i) {
        res[i + 1] += res[i] / BASE;
        res[i] %= BASE;
    }
    while (res.size() > 1 and res.back() == 0) {
        res.pop_back();
    }
    reverse(res.begin(), res.end());
    std::string ab;
    for (int64_t elem : res) {
        ab.push_back('0' + elem);
    }
    return ab;
}
```

Ниже приведён листинг перемножения двух чисел с использованием БПФ.

```
std::string fft_mult(const std::string & a, const std::string & b) {
    size_t max_size = std::max(a.size(), b.size());
    size_t n = 1;
    while (n < max_size) {
        n *= 2;
    }
    n *= 2;
    vc fa(n), fb(n);
    for (size_t i = 0; i < a.size(); ++i) {
        fa[a.size() - i - 1] = complex(a[i] - '0');
    }
    for (size_t i = 0; i < b.size(); ++i) {
        fb[b.size() - i - 1] = complex(b[i] - '0');
    }
    fft(fa, false);
    fft(fb, false);
    for (size_t i = 0; i < n; ++i) {
        fa[i] = fa[i] * fb[i];
    }
```

```
    fft(fa, true);
    vi res(n);
    for (size_t i = 0; i < n; ++i) {
        res[i] = (int64_t)round(fa[i].real());
    }
    for (size_t i = 0; i < n - 1; ++i) {
        res[i + 1] += res[i] / BASE;
        res[i] %= BASE;
    }
    while (res.size() > 1 and res.back() == 0) {
        res.pop_back();
    }
    reverse(res.begin(), res.end());
    std::string ab;
    for (int64_t elem : res) {
        ab.push_back('0' + elem);
    }
    return ab;
}
```

Листинг основной функции:

```
#include "fft_mult.hpp"
#include <iostream>

using namespace std;

int main() {
    string a, b;
    cin >> a >> b;
    string fft_res = fft_mult(a, b);
    cout << fft_res << endl;
}
```

3 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
93401284601794283329
42701674252367504966
$ cat tests/1.out
39883912298184884573526908765418185118l4
$ ./solution <tests/1.in
39883912298184884573526908765418185118l4
$ cat tests/2.in
13008165746621516507460306944292896
31663877276263350780406500557748159
$ cat tests/2.out
41188896379031632091426118289368551880074416330700990597827830077846l
$ ./solution <tests/2.in
41188896379031632091426118289368551880074416330700990597827830077846l
```

4 Тест производительности

В тесте сравниваются перемножение в столбик и с использованием БПФ для чисел с разным числом разрядов.

Количество разрядов	БПФ, мс	В столбик, мс
100	0.973	0.213
200	1.324	0.362
500	1.924	1.862
1000	3.057	7.566
2000	4.555	28.194
5000	18.859	178.206
10^4	37.957	677.440
$2 \cdot 10^4$	80.613	2845.117
$5 \cdot 10^4$	168.944	17920.872
10^5	351.770	-
$2 \cdot 10^5$	746.176	-
$5 \cdot 10^5$	1617.317	-
10^6	3445.852	-

«-» в таблице означает, что замеры времени не производились.

Видно, что при увеличении числа разрядов вдвое, время работы перемножения в столбик возрастает в четыре раза, а время работы перемножения с использованием БПФ примерно в два раза.

5 Выводы

В ходе выполнения курсового проекта я изучил алгоритм быстрого вычисления дискретного преобразования Фурье — быстрое преобразование Фурье.

Основной сложностью было вывести и понять все формулы, которые используются для БПФ. Детали в статьях опускаются, поэтому не всегда очевиден тот или иной переход.

Интересно было узнать, как вычислять БПФ без дополнительной памяти — с помощью так называемого преобразования бабочки, иллюстрации к которому довольно красивые.

В работе перемножаются два числа в столбик и с помощью БПФ. В сравнении видно, насколько медленно перемножение в столбик для сравнительно небольшого числа разрядов. При написании своей длинной арифметики важно использовать БПФ для лучшей производительности.

В течении прошлых лет пытался изучить этот алгоритм, но каждый раз не хватало нескольких деталей для полного понимания. Перечитав больше статей, нашёл необходимые детали. Выбрав эту тему даже появилась идея выложить работу в форме статьи, если другие люди поймут алгоритм, прочитав работу.

Список литературы

[1] *algo :: Быстрое преобразование Фуръе за O (N log N). Применение к умножению двух полиномов или длинных чисел — e-maxx.ru*
URL: `https://e-maxx.ru/algo/fft_multiply` (дата обращения: 08.04.2022).

[2] *Cooley-Tukey FFT algorithm — Wikipedia*
URL: `https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm` (дата обращения: 08.04.2022).