Бързо Преобразуване на Фурие (Fast Fourier Transform) и умножение на дълги числа

Въведение

Бързото преобразуване на Фурие (FFT) намира множество приложения в математиката, физиката и информатиката. В същността си FFT разлага множество от числа по честоти на срещане. В тази статия ще разгледаме основата на FFT във връзка с едно негово приложение-умножението на дълги числа.

Умножение на дълги числа

На информатиците по състезания много често се налага да „си пишат" дългите числа по състезания, заради липсата им като стандартна библиотека. Реализацията на дълго събиране и изваждане е тривиална и следва правилата, по които учат учениците от началното училище да събират и изваждат. Реализацията на дълго умножение по конвенционалния начин също не е трудна за писане, но проблемът е, че сложността й е $O(n^2)$, където n е броят на цифрите на числата (отсега нататък ще шчитаме, че двете числа, които искаме да умножим, са с по n цифри). С развитие на информатиката са намерени много начини да се редуцира асимптотичната сложност на алгоритъма. В тях се прилагат хитри аритметични преобразувания, „разделяй и владей" и много други техники. За повече информация, може да прочетете частта на *[Knuth, The Art Of Computer Programming]*, посветена на бързото умножение на числа.

Чрез FFT се достига асимптотична сложност $O(n \log n)$, която е оптимална за тази операция. Сега да видим как FFT може така драматично да понижи сложността на алгоритъма.

Нека имаме две дълги числа a, b, които са представени в бройна система с основа s с n цифри. Нека приемем, че n е степен на 2(ако това не е изпълнено, добавяме нули отляво на всяко число, докато условието не е изпълнено). Имаме:

$$a = a_0 s^0 + a_1 s^1 + \cdots + a_{n-1} s^{n-1},$$

$$b = b_0 s^0 + b_1 s^1 + \cdots + b_{n-1} s^{n-1}.$$

Всъщност, FFT се прилага в по-общ случай – когато търсим произведение на полиноми, затова да разгледаме съответните полиноми:

$$f_a(x) = a_0 x^0 + a_1 x^1 + \cdots + a_{n-1} x^{n-1},$$

$$f_b(x) = b_0 x^0 + b_1 x^1 + \cdots + b_{n-1} x^{n-1}.$$

За произведението им получаваме:

$$f_{ab}(x) = f_a(x)f_b(x) = \sum_{i=0}^{2n-1} c_i x^i,$$

където:

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}.$$

Тук наблюдението, което ни помага, е че полином от степен n е определен от стойностите, които приема за n различни числа. Полиномът може да се възстанови чрез Лагранжова интерполация. Така, ще имаме достатъчно информация да намерим произведението на два полинома, ако намерим стойностите които приемат те за $2n$ различни числа.

Дотук не сме си помогнали много, защото намирането на тези стойности има квадратна сложност в общия случай. Но някой, който е запознат достатъчно добре с комплексните числа и теорията на числата, ще предложи да използваме степените на корени на единицата, защото така ще се възползваме от връзката им със сравненията. По-формално, ще използваме:

$$\left(\left\{e^{\frac{2\pi ij}{n}}\right\}_{j=0}^{\infty}\right)^{\times} \cong \mathbb{Z}/n\mathbb{Z}.$$

Не се стряскайте от вида на горния израз, той просто казва, че за да умножим два n-ти корена на единицата е достатъчно да съберем две числа по модул n.

Така задачата ни стига до следното: Намери стойностите на

$$a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_{n-1}x^{n-1},$$

където x е n-ти корен на единицата. Тук идва на помощ FFT.

Бързо Преобразуване на Фурие

Бързото преобразуване на Фурие е точно това, което ни трябва: То преобразува множество от комплексни числа $\{x_i\}_{i=0}^{n-1}$ в множство $\{y_i\}_{i=0}^{n-1}$, за което:

$$y_i = \sum_{j=0}^{n-1} x_i e^{\frac{2\pi ij}{n}} = f\left(e^{\frac{2\pi i}{n}}\right) = \sum_{j=0}^{n-1} x_i w^{ij},$$

където:

$$f(t) = x_0t^0 + x_1t^1 + \cdots + x_{n-1}t^{n-1}.$$

За ефективно прилагане на FFT използваме следните зависимости(тук $\omega = e^{\frac{2\pi i}{n}}$):

$$\omega^n = 1,$$

$$\omega^{n+k} = \omega^k,$$

$$\omega^{\frac{n}{2}} = -1,$$

$$\omega^{\frac{n}{2}+k} = -\omega^k.$$

Сега можем да приложим „разделяй и владей“: разлагаме полинома на два:

$$f_0(t) = x_0 t^0 + x_2 t^1 + x_4 t^2 + \cdots + x_{n-2} t^{\frac{n}{2}-1},$$

$$f_1(t) = x_1 t^0 + x_3 t^1 + x_5 t^2 + \cdots + x_{n-1} t^{\frac{n}{2}-1},$$

$$f(t) = f_0(t) + t f_1(t).$$

Изчисляваме FFT на двата полинома, после сливаме, за да получим FFT на началния полином, като използваме правилата за смятане с корени на единицата.

Примерен код – C++

```
#include <complex>
#include <cmath>
#include <iostream>
using namespace std;

typedef double num;
typedef complex<double> cnum;
const double PI = 3.141592654;

void FFT(cnum* a, int n){
	int i;
	cnum *a0, * a1, u(1,0), un = exp(cnum(0,2.0*PI)/cnum(n));
	if(n == 1) return;
	a0 = new cnum[n/2];
	a1 = new cnum[n/2];
	for(i = 0; i < n/2; i++){
		a0[i] = a[2*i];
		a1[i] = a[2*i+1];
	}
	FFT(a0, n/2);
	FFT(a1, n/2);
	for(i = 0; i < n/2; i++){
		a[i] = a0[i] + u*a1[i];
		a[i+n/2] = a0[i] - u*a1[i];
		u*=un;
	}
}


int main(){
	int i;
	cnum a[4];
	a[0] = cnum(1,0);
```

```
        a[1] = cnum(2,0);
        a[2] = cnum(3,0);
        a[3] = cnum(4,0);
        FFT(a, 4);
        for(i = 0; i < 4; i++) cout << a[i] << ' ';
        cout << endl;
        system("pause");
        return 0;
}
```

Примерен код – Mathematica

```
FFT[a_] := Module[{a0, a1, n = Length@a, r0, r1},
  If[n == 1, Return@a];
  a0 = FFT@Pick[a, OddQ /@ Range[n]];
  a1 = FFT@Pick[a, EvenQ /@ Range[n]];
  r0 = (a0[[#]] + (E^(2.0 Pi I # / n)) a1[[#]]) & /@ Range[n / 2];
  r1 = (a0[[#]] - (E^(2.0 Pi I # / n)) a1[[#]]) & /@ Range[n / 2];
  Return@Join[r0, r1];
 ]
```

Обратно Преобразуване на Фурие

За да има смисъл прилагането на бързото преобразуване на Фурие във нашия случай ни е необходимо бързо да извършваме обратната трансформация: ако знаем стойностите на полином от n-та степен за n-те корена на единицата, как бързо да възстановим коефициентите на съответния полином. Обратното преобразувение на Фурие се дава с формулата:

$$x_i = \frac{1}{n}\sum_{j=0}^{n-1} y_j w^{-ij}$$

Формулите за правото и обратното преобразуване си приличат достатъчно за да можем да приложим същата идея разделяй и владей и за обратното преобразуване.

Умножение на дълги числа

Умножението на дълги числа чрез FFT се основава на следната зависимост:

$$\mathcal{F}(A * B) = \mathcal{F}(A).\mathcal{F}(B)$$

Тук „*” е обикновенно умножение (конволюция), а „.” Е скаларно умножение(което се пресмята със линейна сложност).

Сега алгоритъма за умножение на две числа е ясен: прибавяме n нули отляво на всяко число, прилагаме FFT на двете редици от цифри, после поелементно умножаваме получените редици, и към резултата прилагаме обратното преобразуване.

Примерен код – C++

```
//Multiplication of big integers
//Based on Fast Fourier Transform
//by krassi_holmz, 2009

#include <iostream>
#include <complex>
#include <cmath>
using namespace std;

typedef double num;
typedef complex<num> cnum;
const double Pi = 3.14159265358979;

const int max_n = (1<<10);

void FFT(cnum* a, int n){
	int i;
	cnum* a0, * a1, w(1,0), wn =
exp(cnum(0.0,2.0*Pi)/cnum(n));
	if(n == 1) return;
	a0 = new cnum[n/2];
	a1 = new cnum[n/2];
	for(i = 0; i < n/2; i++){
		a0[i] = a[2*i];
		a1[i] = a[2*i+1];
	}
	FFT(a0,n/2);
	FFT(a1,n/2);
	for(i = 0; i < n/2; i++){
		a[i] = a0[i] + w*a1[i];
		a[i+n/2] = a0[i] - w*a1[i];
		w*=wn;
	}
}

void IFFT(cnum* a, int n, bool scale){
	int i;
	cnum* a0, * a1, w(1,0), wn = exp(cnum(0.0,2.0*Pi*(n-
1))/cnum(n));
	if(n == 1) return;
	a0 = new cnum[n/2];
	a1 = new cnum[n/2];
	for(i = 0; i < n/2; i++){
		a0[i] = a[2*i];
		a1[i] = a[2*i+1];
	}
	IFFT(a0,n/2,false);
	IFFT(a1,n/2,false);
	for(i = 0; i < n/2; i++){
```

```
            a[i] = a0[i] + w*a1[i];
            a[i+n/2] = a0[i] - w*a1[i];
            w*=wn;
      }
      if(scale)
            for(i = 0; i < n; i++)
                  a[i] /= cnum(n);
}

void FFTmult(int* a, int* b, int* c, int n){
      int i, t;
      cnum* ca, * cb, * cc;
      ca = new cnum[2*n];
      cb = new cnum[2*n];
      cc = new cnum[2*n];
      for(i = 0; i < n; i++){
            ca[i] = cnum(a[i],0);
            cb[i] = cnum(b[i],0);
            ca[i+n] = cnum(0,0);
            cb[i+n] = cnum(0,0);
      }
      FFT(ca, 2*n);
      FFT(cb, 2*n);
      for(i = 0; i < 2*n; i++)
            cc[i] = ca[i]*cb[i];
      IFFT(cc, 2*n, true);
      for(i = 0; i < 2*n; i++)
            c[i] = (int)floor(cc[i].real()+0.5);
      t = 0;
      for(i = 0; i < 2*n; i++){
            c[i] += t;
            t = c[i]/10;
            c[i] %= 10;
      }
}

int a[max_n], b[max_n], c[max_n];

void read(int* p){
      int i = 0, j;
      char ch;
      for(;;){
            ch = getc(stdin);
            if(ch == '\n') break;
            p[i++] = ch - '0';
      }
      for(j = 0; j < i/2; j++)
            swap(p[j], p[i-j-1]);
}

int main(){
      int i;
      read(a);
      read(b);
      FFTmult(a, b, c, max_n/2);
      i = max_n - 1;
      while(!c[i]) i--;
```

```
	for(;i>=0;i--)
		printf("%d", c[i]);
	printf("\n");
	system("pause");
	return 0;
}
```