Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

**Дискретное преобразование Фурье**

Лабораторная работа №2

студента 5 курса 531 группы

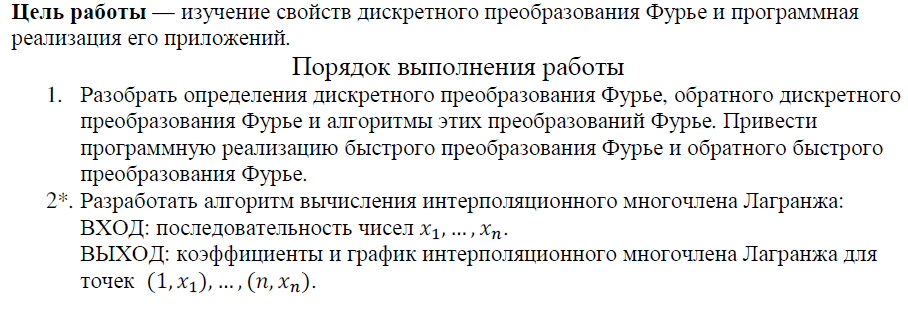
специальности 10.05.01 «Компьютерная безопасность»

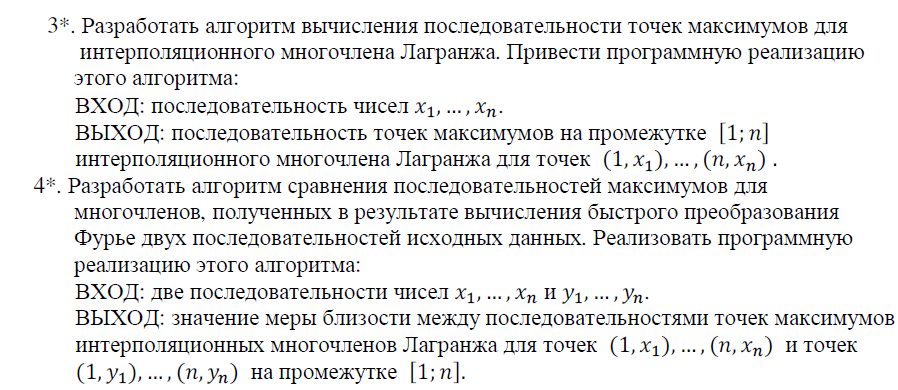
факультета компьютерных наук и информационных технологий

Завенягина Максима Павловича

|  |  |  |
| --- | --- | --- |
| Преподаватель:  Профессор, д. ф-м. н. | \_\_\_\_\_\_\_\_\_\_ | В. А. Молчанов |
|  | подпись, дата |  |

Саратов 2018





**Быстрое преобразование Фурье и умножение полиномов**

Пусть имеется многочлен *n*-ой степени:

Не теряя общности, можно считать, что *n* является степенью 2. Если в действительности *n* не является степенью 2, то мы просто добавим недостающие коэффициенты, положив их равными нулю.

Из теории функций комплексного переменного известно, что комплексных корней *n*-ой степени из единицы существует ровно *n*. Обозначим эти корни через , тогда известно, что  . Кроме того, один из этих корней  (называемый главным значением корня *n*-ой степени из единицы) таков, что все остальные корни являются его степенями: .

Тогда дискретным преобразованием Фурье (ДПФ) (discrete Fourier transform, DFT) многочлена *A(x)* называются значения этого многочлена в точках , т.е. это вектор:

Аналогично определяется и обратное дискретное преобразование Фурье (InverseDFT). Обратное ДПФ для вектора значений многочлена  – это вектор коэффициентов многочлена :

Таким образом, если прямое ДПФ переходит от коэффициентов многочлена к его значениям в комплексных корнях *n*-ой степени из единицы, то обратное ДПФ — наоборот, по значениям многочлена восстанавливает коэффициенты многочлена.

**Быстрое преобразование Фурье** (fast Fourier transform) — это метод, позволяющий вычислять ДПФ за время *O*(*nlogn*). Этот метод основывается на свойствах комплексных корней из единицы (а именно, на том, что степени одних корней дают другие корни).

Основная идея БПФ заключается в разделении вектора коэффициентов на два вектора, рекурсивном вычислении ДПФ для них, и объединении результатов в одно БПФ.

Итак, пусть имеется многочлен *A(x)* степени *n*, где *n* — степень двойки, и *n>*1:

Разделим его на два многочлена, один — с чётными, а другой — с нечётными коэффициентами:

Нетрудно убедиться, что:

Многочлены *A0* и *A1* имеют вдвое меньшую степень, чем многочлен *A*. Если мы сможем за линейное время по вычисленным DFT(*A0* ) и DFT(*A1)* вычислить DFT(*A*), то мы и получим искомый алгоритм быстрого преобразования Фурье (т.к. это стандартная схема алгоритма "разделяй и властвуй", и для неё известна асимптотическая оценка *O*(*nlogn*)).

Итак, пусть мы имеем вычисленные вектора  и  . Найдём выражения для .

Во-первых, вспоминая (1), мы сразу получаем значения для первой половины коэффициентов:

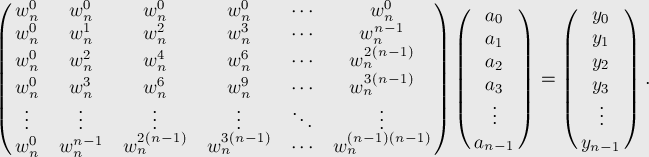
Для второй половины коэффициентов после преобразований также получаем простую формулу:

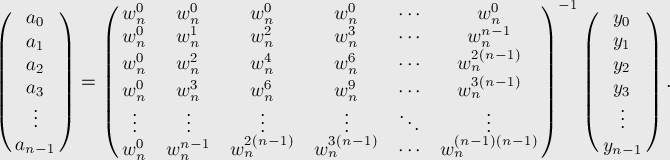
(Здесь мы воспользовались (1), а также тождествами   и  .)

Итак, в результате мы получили формулы для вычисления всего вектора {*yk*}:

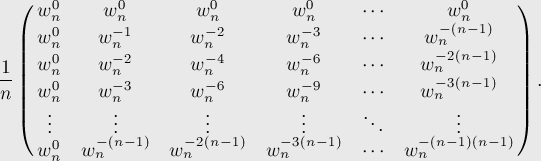
**Обратное БПФ**

Итак, пусть дан вектор  — значения многочлена *A* степени *n* в точках . Требуется восстановить коэффициенты  многочлена. Эта задача называется **интерполяцией**, для этой задачи есть и общие алгоритмы решения, однако в данном случае будет получен очень простой алгоритм (простой тем, что он практически не отличается от прямого БПФ).

ДПФ мы можем записать, согласно его определению, в матричном виде:

Тогда вектор  можно найти, умножив вектор  на обратную матрицу к матрице, стоящей слева (которая, кстати, называется матрицей Вандермонда):

Непосредственной проверкой можно убедиться в том, что эта обратная матрица такова:



Таким образом, получаем формулу:

 a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}.[...]

Сравнивая её с формулой для y_k:

 y_k = \sum_{j=0}^{n-1} a_j w_n^{kj}, 

Можно заметить, что эти две задачи почти ничем не отличаются, поэтому коэффициенты *ak* можно находить таким же алгоритмом "разделяй и властвуй", как и прямое БПФ, только вместо  везде надо использовать , а каждый элемент результата надо разделить на *n*.

Таким образом, вычисление обратного ДПФ почти не отличается от вычисления прямого ДПФ, и его также можно выполнять за время *O*(*nlogn)*

**Алгоритм для прямого и обратного быстрого преобразования Фурье над**

typedef complex<double> base;

void fft (vector<base> & a, bool invert) {

int n = (int) a.size();

if (n == 1) return;

vector<base> a0 (n/2), a1 (n/2);

for (int i=0, j=0; i<n; i+=2, ++j) {

a0[j] = a[i];

a1[j] = a[i+1];

}

fft (a0, invert);

fft (a1, invert);

double ang = 2\*PI/n \* (invert ? -1 : 1);

base w (1), wn (cos(ang), sin(ang));

for (int i=0; i<n/2; ++i) {

a[i] = a0[i] + w \* a1[i];

a[i+n/2] = a0[i] - w \* a1[i];

if (invert)

a[i] /= 2, a[i+n/2] /= 2;

w \*= wn;

}

}

Результат будет лежать в векторе комплексных чисел, поданном на вход функции по ссылке. Переменная invert определяет, прямое или обратное преобразование нужно применить.

**Алгоритм для прямого и обратного быстрого преобразования Фурье над**

Вход:

Выход:

Для применения Быстрого преобразования Фурье нам нужно, чтобы примитивный корень существовал для некоторого n, являвшегося степенью двойки, а также всех меньших степеней. И если в комплексном случае примитивный корень существовал для любого n, то в случае модульной арифметики это, вообще говоря, не так, поэтому введем ограничение: пусть делит .

void fft (vector<int> & a, bool invert, int mod) {

int n = (int) a.size();

if (n == 1) return;

vector< int > a0 (n/2), a1 (n/2);

for (int i=0, j=0; i<n; i+=2, ++j) {

a0[j] = a[i];

a1[j] = a[i+1];

}

fft (a0, invert, mod);

fft (a1, invert, mod);

int root = -1;

for (root = 2; ; root++)

{

if (root.PowOnMod(n, mod) != 1)

continue;

bool f = false;

for (int i = 1; i < n; i++)

{

if (root.PowOnMod(i, mod) == 1)

{

f = true;

break;

}

}

if (!f)

break;

}

int w = 1, wn = invert ? root.GetInverse(mod) : root;

for (int i=0; i<n/2; ++i) {

a[i] = a0[i] + w \* a1[i];

a[i+n/2] = a0[i] - w \* a1[i];

if (invert)

a[i] /= 2, a[i+n/2] /= 2;

w \*= wn;

}

}

Тут функция PowOnMod возводит число в степень по модулю, а функция GetInverse находит обратный элемент в поле с помощью расширенного алгоритма Евклида. Также стоит заметить, что под всеми операциями подразумеваются переопределенные операции над полем.

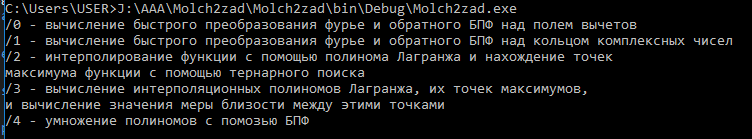
Стоит отметить, что приведенные алгоритмы вычислений БПФ могут быть также реализованы не рекурсивно, что позволит улучшить производительность (хотя асимптотика алгоритмов сохранится).

**Алгоритм умножения полиномов с помощью быстрого преобразования Фурье**

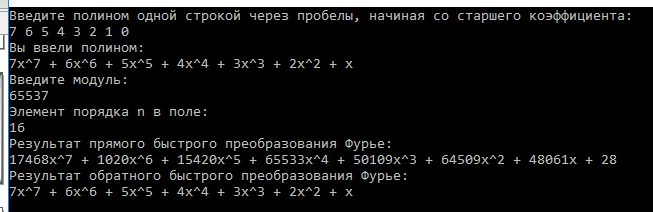
С помощью БПФ можно немного улучшить скорость умножения полиномов. Пусть даны два полинома и . и – это два вектора – значения многочленов. При умножении многочленом получается многочлен со значениями, равными произведениям значений, полученных при применении и в соответствующих точках. Из этого следует, что, чтобы перемножить 2 полинома, достаточно применить к ним прямое БПФ, перемножить коэффициенты с одинаковыми индексами, после чего применить обратное БФП к результату. Алгоритм перемножения двух многочленов с помощью БПФ очевиден.

**Пример работы программы**

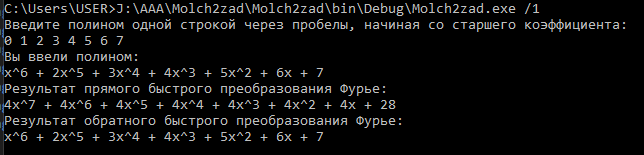
Программа имеет один обязательный параметр. Запуск без параметров укажет на это:



Пример быстрого преобразования Фурье над полем:

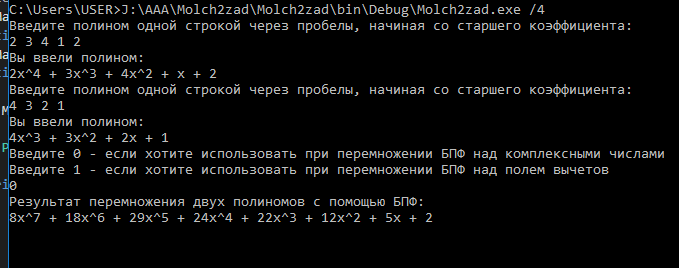
****

Пример быстрого преобразования Фурье над :

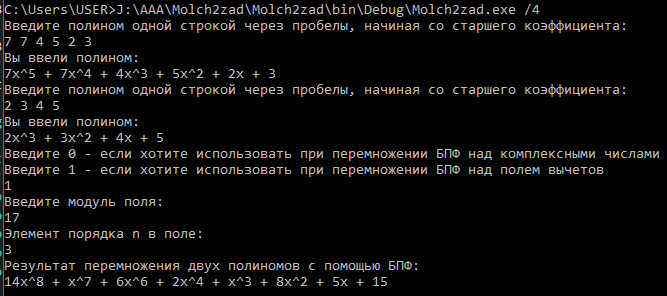


Тут стоит отметить, что в результате прямого быстрого преобразования могут получиться числа со значащей мнимой частью, но для удобства в интерфейсе эта часть не отображается.

Пример умножения полиномов над с помощью быстрого преобразования Фурье:



Пример умножения полиномов над полем с помощью быстрого преобразования Фурье:



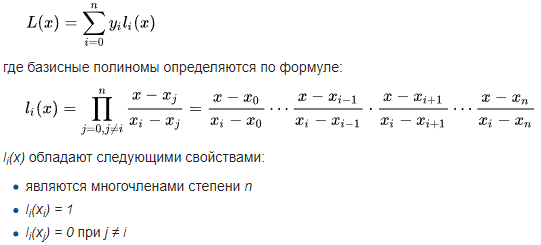
Как видно на рисунке, коэффициенты в полученном многочлене также принадлежат полю.

**Вычисление интерполяционного многочлена Лагранжа и последовательности точек максимумов для него**

**Интерполяцио́нный многочле́н Лагра́нжа** —[многочлен](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D1%87%D0%BB%D0%B5%D0%BD) минимальной степени, принимающий данные значения в данном наборе точек. Для *n+1* пар чисел (*x0*, *y0*), (*x1*, *y1*),…, (*xn*, *yn*), где все *xj* различны, существует единственный многочлен *L(x)* степени не более *n*, для которого *L(xj) = yj*.

В простейшем случае (*n=1*) — это линейный многочлен, [график](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D0%BA_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8) которого — прямая, проходящая через две заданные точки.

Лагранж предложил способ вычисления таких многочленов:



**Алгоритм:**

List<Polinom> lp = new List<Polinom>();

for (int i = 0; i < points.Count; i++)

{

List<double> ld = new List<double> { 1 };

Polinom p = new Polinom(ld);

for (int j = 0; j < points.Count; j++)

{

if (i == j)

continue;

List<double> ld2 = new List<double>

{

(-points[j].Key) / (points[i].Key - points[j].Key),

((double)1) / (points[i].Key - points[j].Key)

};

p = p \* (new Polinom(ld2));

}

lp.Add(p);

}

List<double> ld\_res = new List<double>();

Polinom res = new Polinom(ld\_res);

for (int i = 0; i < lp.Count; i++)

res = res + lp[i] \* points[i].Value;

Где Polinom – это класс, для объектов которого переопределены соответствующие операции.

**Нахождение последовательности точек максимумов.**

Стоит отметить, что в результате интерполирования значений, получается такой полином, что функция, реализуемая этим полиномом, меняет свою производную на участках не более одного раза, из чего следует, что функция на каждом из таких отрезков является унимодальной.

Непрерывная функция   называется унимодальной на отрезке ,если:  
1. Точка локального минимума функции принадлежит отрезку ;  
2. для любых двух точек отрезка , взятых по одну сторону от точки минимума, точке, более близкой к точке минимума соответствует меньшее значение функции; то есть из условий или cледует условие .

На унимодальных функциях применим алгоритм тернарного поиска:

while (r - l > eps)

{

double m1 = l + (r - l) / 3,

m2 = r - (r - l) / 3;

if (Calc(p, m1) < Calc(p, m2))

l = m1;

else

r = m2;

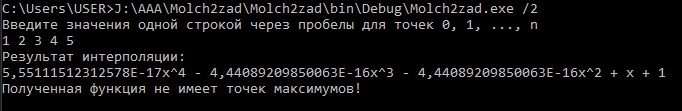
}

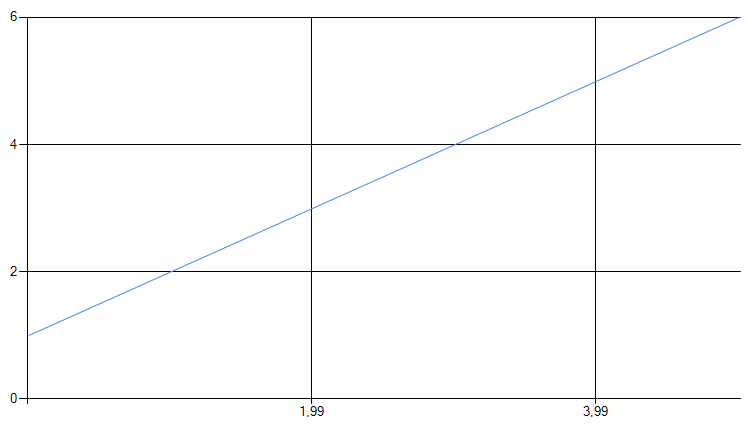
Тут eps – число, характеризующее точность вычисления, функция Calc – вычисляет значение многочлена в точке, p – объект класса Polinom, l и r – левая и правая границы отрезка соответственно.

Также стоит отметить, что точки m1 и m2 не обязательно должны делить отрезок на 3 равных части. Эти точки могут располагаться как угодно при условии, что m1 < m2.

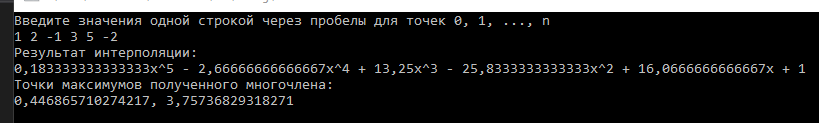
**Пример работы программы:**

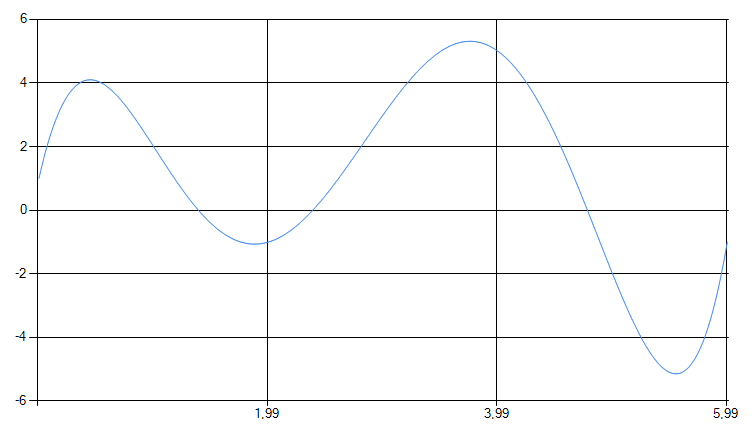
Пример 1:





Пример 2:





**Сравнение последовательностей точек максимумов, полученных в результате интерполирования**

**Алгоритм:**

int b = 0;

List<double> res = new List<double>();

for (int i = 0; i < ld1.Count; i++)

{

while (b < ld2.Count - 1 && (Math.Abs(ld1[i] - ld2[b]) > Math.Abs(ld1[i] - ld2[b + 1])))

b++;

res.Add(Math.Abs(ld1[i] - ld2[b]));

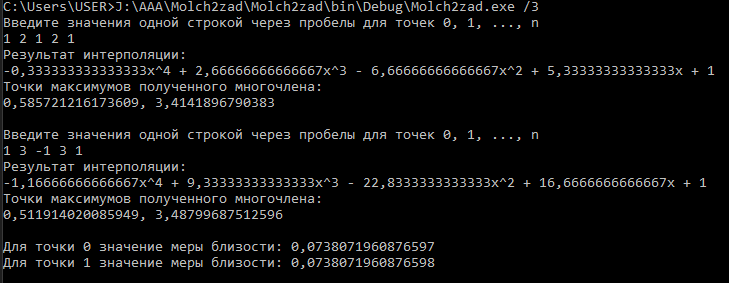
}

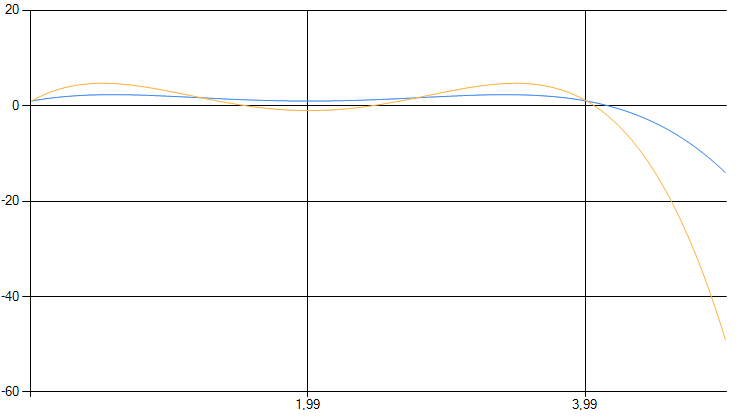
return res;

Тут ld1 и ld2 – массивы точек максимумов первого и второго полиномов соответственно.

**Пример работы программы:**

Пример 1:





Пример 2:

