B-Tech Project AUG-NOV 2023

Implementation of number theoretical algorithms

Aakash
EP21BTECH11001

Under the guidance of

Dr.Rogers Mathew

Co-supervisor
Dr Shantanu Desai

DEPARTMENT OF PHYSICS

INDIAN INSTITUTE OF TECHNOLOGY HYDERABAD

Department of Physics
Indian Institute of Technology
Hyderabad
India

DECLARATION

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and not have misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

Aakash
EP21BTECH11001

Department of Department of Physics
Indian Institute of Technology
Hyderabad
India

ACKNOWLEDGEMENTS

I express my deep gratitude to Dr. Rogers Mathew for his constant guidance andsupport. I have learned a lot from this project and I can't thank him enough.

References [1] Sisper, M. (2013). Introduction to the Theory of Computation (3rd ed.)

Department of Physics
Indian Institute of Technology
Hyderabad
India

APPROVAL SHEET

This report titled “Implementation of number theoretical algorithms” is submitted by Aakash as a partof his BTech project.

Dr. Rogers Mathew
(Supervisor)

ABSTRACT

This research paper delves into significant advancements in algorithmic theory, exploring four key areas that have played pivotal roles in computational complexity and problem-solving strategies. The paper investigates the Miller-Rabin primality test, the AKS primality test, Fast Fourier Transform (FFT) for polynomial multiplication, and the concept of NP-completeness.

Firstly, the Miller-Rabin primality test, a probabilistic algorithm, is examined for its efficiency in determining the primality of large numbers. We explore its probabilistic nature, its applications, and its trade-offs between accuracy and computational complexity.

Secondly ,Fast Fourier Transform (FFT) algorithm as applied to polynomial multiplication. We explore how FFT, an algorithm designed for efficient computation of the Discrete Fourier Transform, has been adapted to revolutionize the multiplication of polynomials, resulting in significant improvements in time complexity compared to traditional methods.

Thirdly, we will see the AKS primality test and it's code.

Lastly, the paper delves into the theoretical foundations of NP-completeness, a critical concept in the field of computational complexity theory. It explores the implications of NP-completeness in the classification of computational problems, shedding light on the inherent difficulty of certain problems and the implications for the broader field of algorithms.

Contents

Miller Rabin's Test

Ordinary Prime number checking algorithm take O(n^(1/2)). But miller rabin's Introduced a algorithm that can test a number is prime is or not in O(k*log(n)) Time complexity. Here k is factor for accuracy.
In miller rabin's fermat little theorem is used.
Fermat's Little theorem states that if p is a prime number and a is an interger not divisible by p, then a raised to power p-1 give 1 as remainder when modulo by p.

$$a^{p-1} = 1 \pmod{p}$$

If p is prime.

For an odd integer $n > 1$, factor out the largest power of 2 from $n - 1$, say $n - 1 = 2^e k$ where $e \geq 1$ and k is odd. This meaning for e and k will be used throughout. The polynomial $x^{n-1} - 1 = x^{2^e k} - 1$ can be factored repeatedly as often as we have powers of 2 in the exponent:

$$\begin{aligned} x^{2^e k} - 1 &= (x^{2^{e-1} k})^2 - 1 \\ &= (x^{2^{e-1} k} - 1)((x^{2^{e-1} k} + 1) \\ &= (x^{2^{e-2} k} - 1)(x^{2^{e-2} k} + 1)((x^{2^{e-1} k} + 1) \\ &\vdots \\ &= (x^k - 1)(x^k + 1)(x^{2k} + 1)(x^{4k} + 1) \cdots (x^{2^{e-1} k} + 1). \end{aligned}$$

If n is prime and $1 \leq a \leq n - 1$ then $a^{n-1} - 1 \equiv 0 \bmod n$ by Fermat's little theorem, so using the above factorization we have

$$(a^k - 1)(a^k + 1)(a^{2k} + 1)(a^{4k} + 1) \cdots (a^{2^{e-1} k} + 1) \equiv 0 \bmod n.$$

When n is prime one of these factors must be $0 \bmod n$, so

$$a^k \equiv 1 \bmod n \text{ or } a^{2^i k} \equiv -1 \bmod n \text{ for some } i \in \{0, \ldots, e - 1\}. \tag{2.1}$$

```java
import java.util.*;
public class miller_rabin_test {
    public static void main(String[] args) {
        int k = 5; // for accuracy

        System.out.println("All primes smaller "
                + "than 100: ");
        ArrayList<Integer> list=new ArrayList<>();
        for (int n = 1; n < 100; n++) {
            if (isPrime(n, k)) list.add(n);
        }
        //if (isPrime(67, k)) list.add(67);
        System.out.println(list);
        System.out.println(list.size());
    }
    static boolean isPrime(int n,int k) {
        if(n<=1) return false;
        if(n<=3) return true;
        if(n%2==0) return false;
        int p=n-1;
        while((p&1)==0) {
            p=p/2;
        }
        for(int i=0;i<k;i++) {
            if(!millerTest(n,p)) return false;
        }
        return true;
    }
    static boolean millerTest(int n,int p) {
        int a=2+(int)(Math.random()%(n-4));
        long T=init(a,p,n);
        if(T==1||T==n-1) return true;
        while(p!=n-1) {
            T=(T*T)%n;
            p*=2;
            //if(T==1) return false;
            if(T==(n-1)) return true;
        }
        return false;
    }
    static long init(int a,int p,int n) {
        long T=1;
        long m=a;
        while(p!=0) {
            if(((p&1)==1)) {
                T*=a;
                T%=n;
            }
            a*=a;
            a%=n;
            p=p>>1;
        }
        return T;
    }

}
```

In the above code we are just taking a number and checking that it is prime or not by miller rabin's test.
It returns false if n is composite and return true if n is probably prime. Here k is a factor for accuracy. Higher the value k greater it's accuracy.

Here:

Boolean isPrime(int n,int k)

1. It handle base case n<4
2. If n is even return false;
3. P=n-1 as n will be odd so p will guaranted be odd
4. Then find a odd number p by dividing p by 2 until it become odd.
5. Calling the millerTest(n,p) function k times:
6. If it returns false then p is definitely composite else it can prime
7. At last if K times millerTest function give true then return true it can be composite also bcz this will not give accurate answer but for large value of K it's accuracy is nearly 100%.

```
Boolean millerTest(int n, int d)
1) Pick a random number 'a' in range [2, n-2]
2) Compute: T = pow(a, p) % n
3) If T == 1 or T == n-1, return true.
4) Do following while d doesn't become n-1.
     T = (T*T) % n
     If (T == n-1) return true
```

```
So here no extra space is used.
And for K times the complexity is O(log(n))
```

So
Time Complexity: O(k*logn)
Auxiliary Space: O(1)

Polynomial Multiplication:

Naïve Approach:
Naïve approach for poynomial multiplication approach takes O(n^2) time complexity.

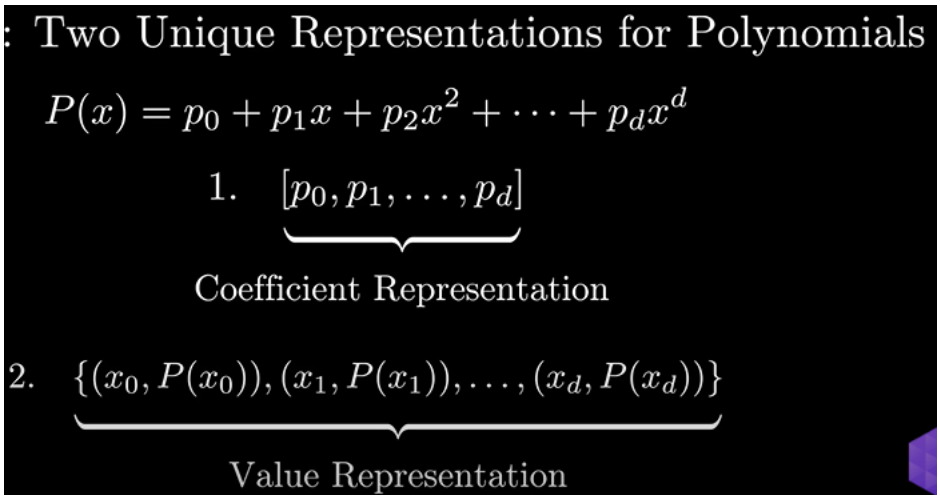
```java
import java.util.*;
public class naive_polynomial_multiplication {
    public static void main(String[] args) {
        int[] a=new int[]{1,2,3};
        int[] b=new int[]{2,3,4};
        System.out.println(func1(a,b));
    }
    static ArrayList<Integer> func1(int[] a,int[] b) {
        ArrayList<Integer> list=new ArrayList<>();
        int n=a.length;
        n*=2;
        n--;
        for(int i=0;i<n;i++) {
            list.add(0);
        }
        for(int i=0;i<a.length;i++) {
            for(int j=0;j<b.length;j++) {
                list.set(i+j,list.get(i+j)+a[i]*b[j]);
            }
        }
        return list;
    }
}
```

Time complexity: $O(n^2)$
Auxiliary space: O(n) as space taken is constant.

By using FFT we can do it in a complexity of O(nlog(n))

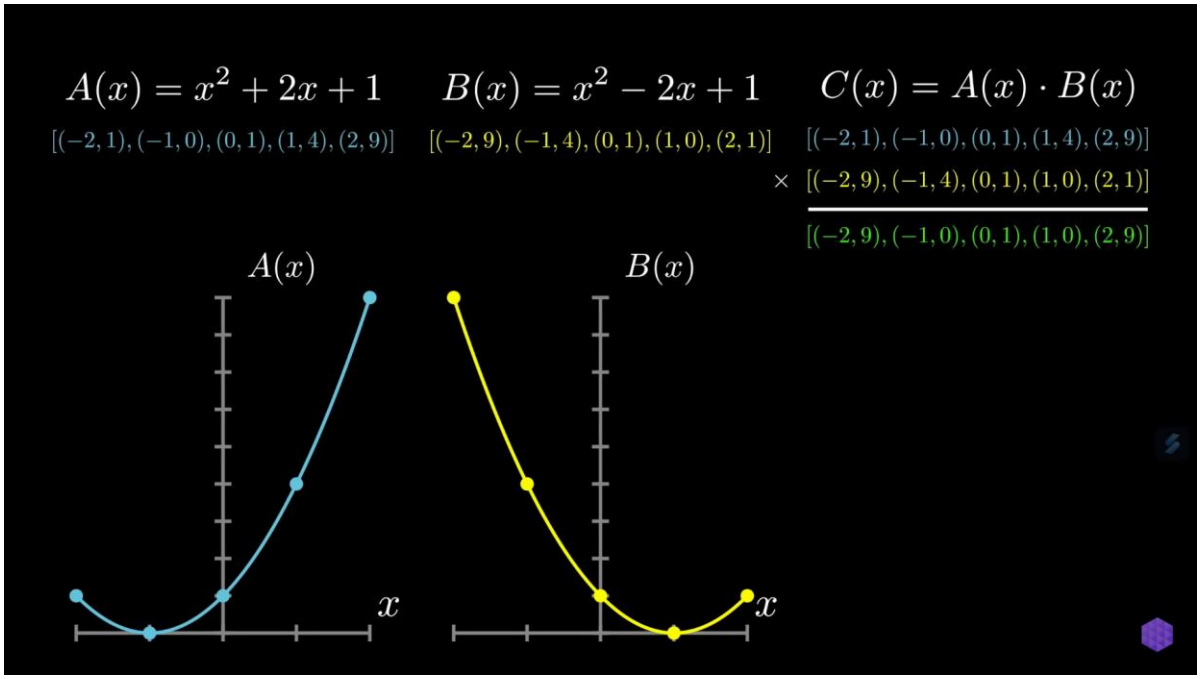
Polynomial multiplication by FFT(Fast Fourier Transfrom):

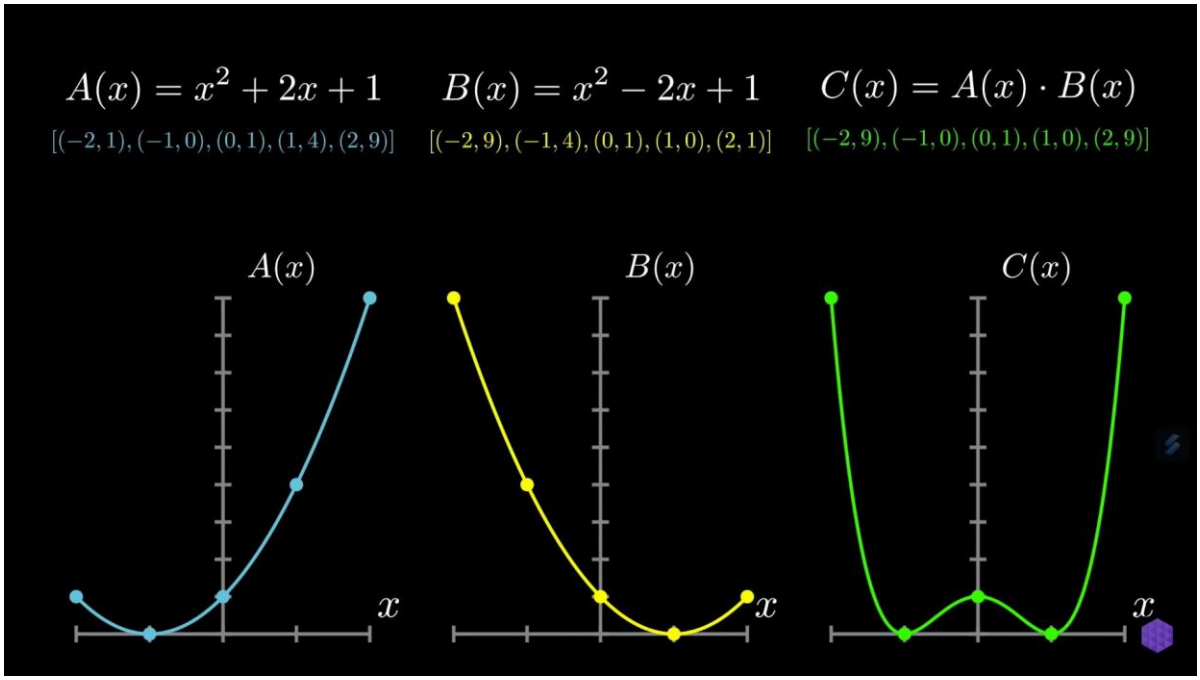
As we know that a polynomial can be represented in point form.
A polynomial of n degree can be represented by n+1 point.
i.e.
suppose a polynomial y=x+3
it can be represented by 2 point as degree is 1
let us say P1(0,3) and P3(1,4)
similarly if 2 degree polynomial then
it can be represented by 3 point
similarly if n degree polynomial then
it can be represented by n+1 point

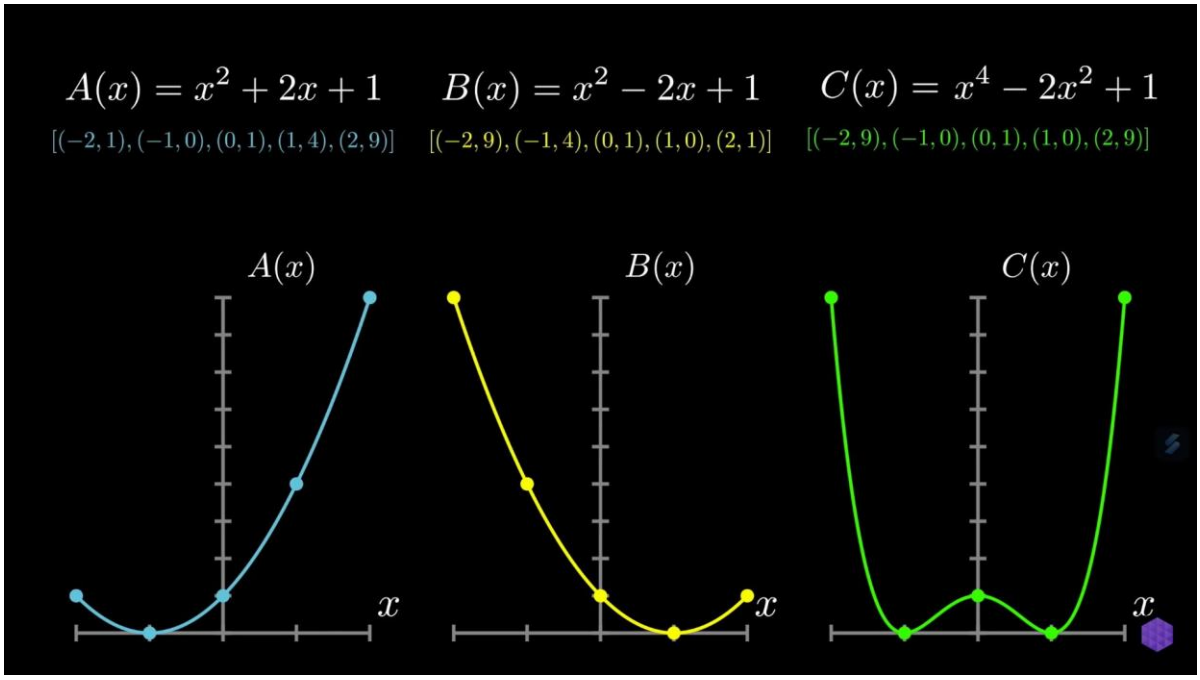


we can get multiplied polynomial point form two polynomial from the give polynomial point form in O(N) time complexity.

Now let us see how it works
Let us take two polynomial A and B







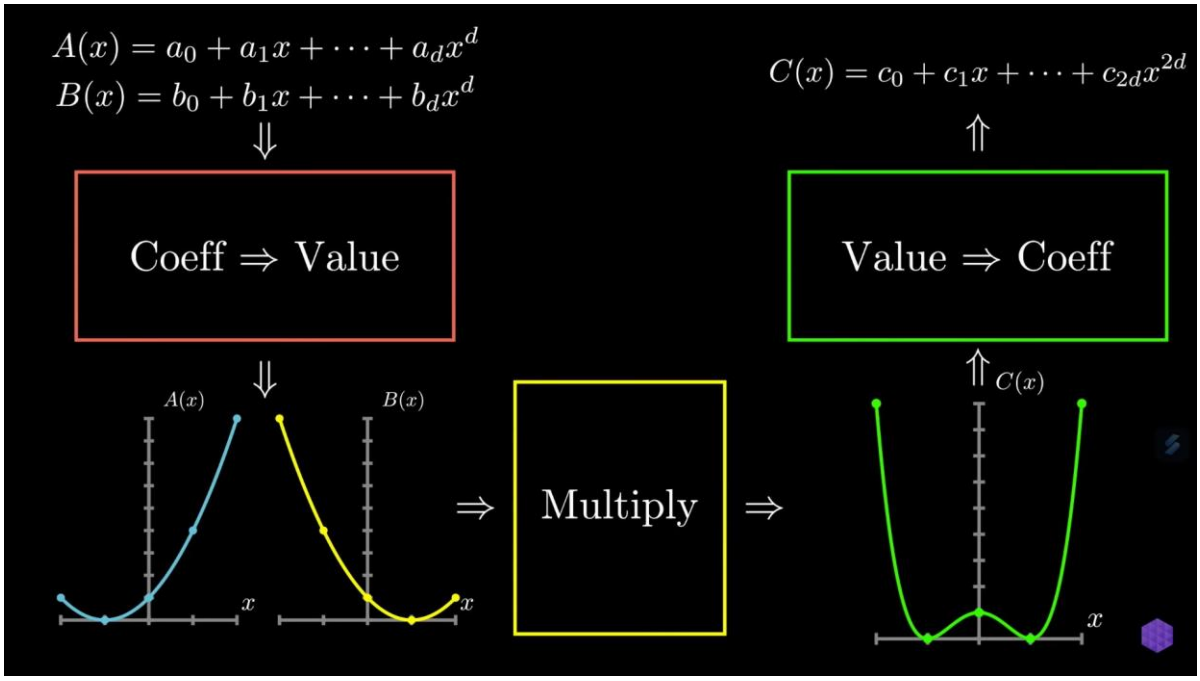
So as we get point to multiplied point form.
But we also need to convert polynomial to point form And also from multiplied point form to required polynomial.

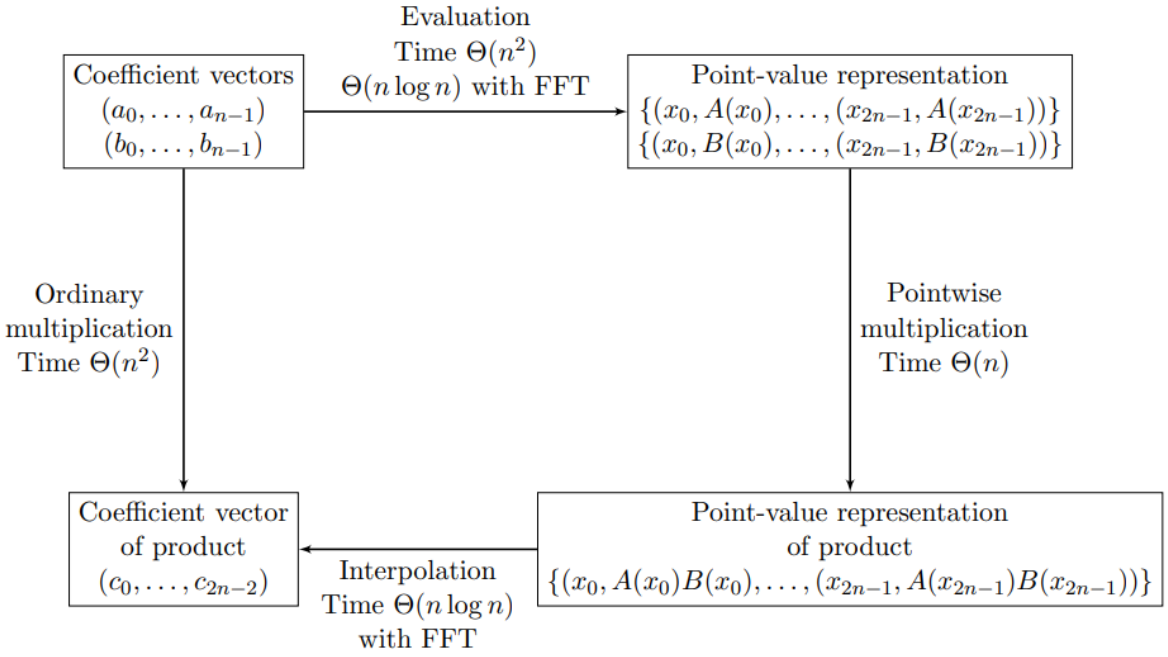
So there are 3 steps to do polynomial multiplication:

Step1:- **Polynomial to point form using FFT:**
For the given polynomial P1 and P2 using FFT convert Polynomial P1 and P2 to point form.

Step 2**:- Point form to Multiplied point form:**
After getting P1 and P2 in point form get P1*P2(multiplied point form).

Step3:- **Multiplied point form to Polynomial form:**
As P1*P2 is in point form, so now using IFFT to convert P1*P2 from point form to polynomial form.



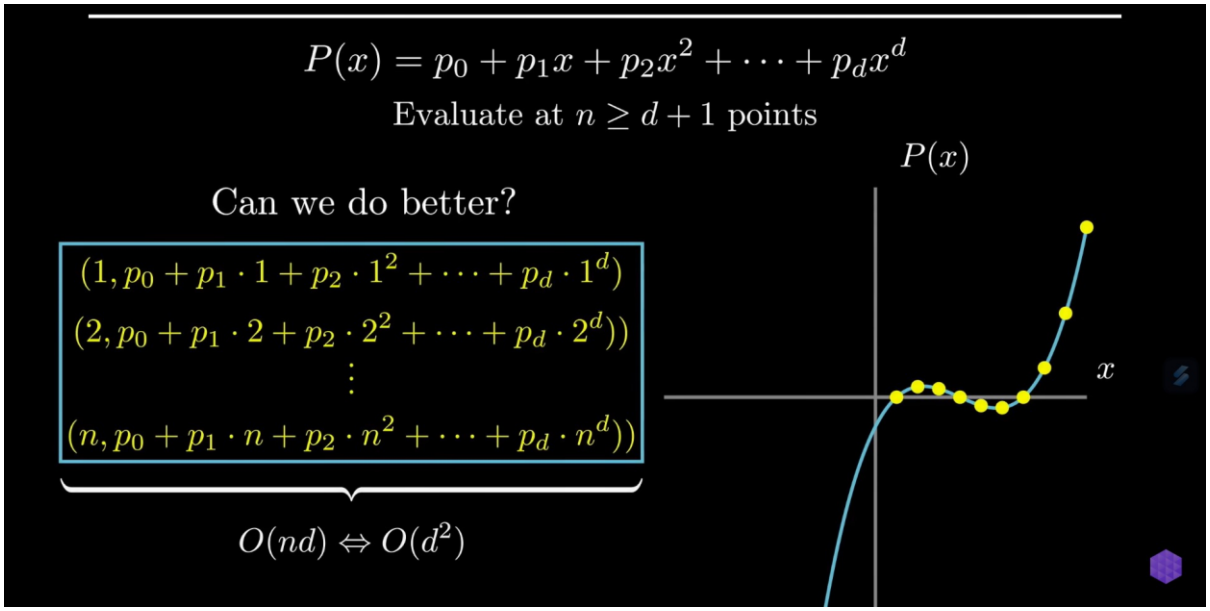


So as here we see Ordinary Time Complexity is O(n^2)
But using FFT Time Complexity = nlogn+ n+ nlogn

Overall it is O(nlogn) time complexity
Now let us understand each step

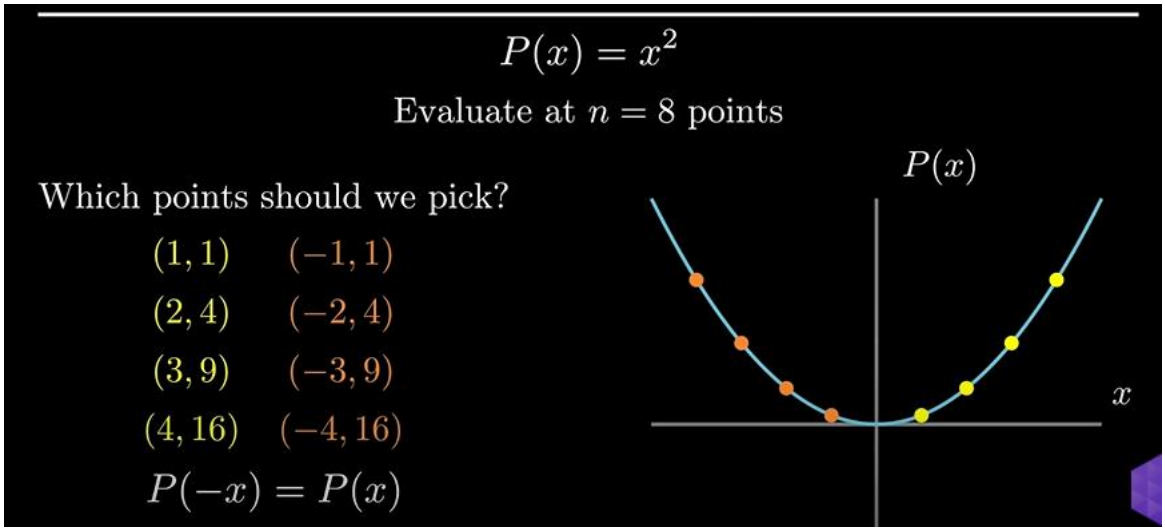
Step1: **Polynomial to point form using FFT:**

So to convert polynomial form to point form if we use naïve approach.

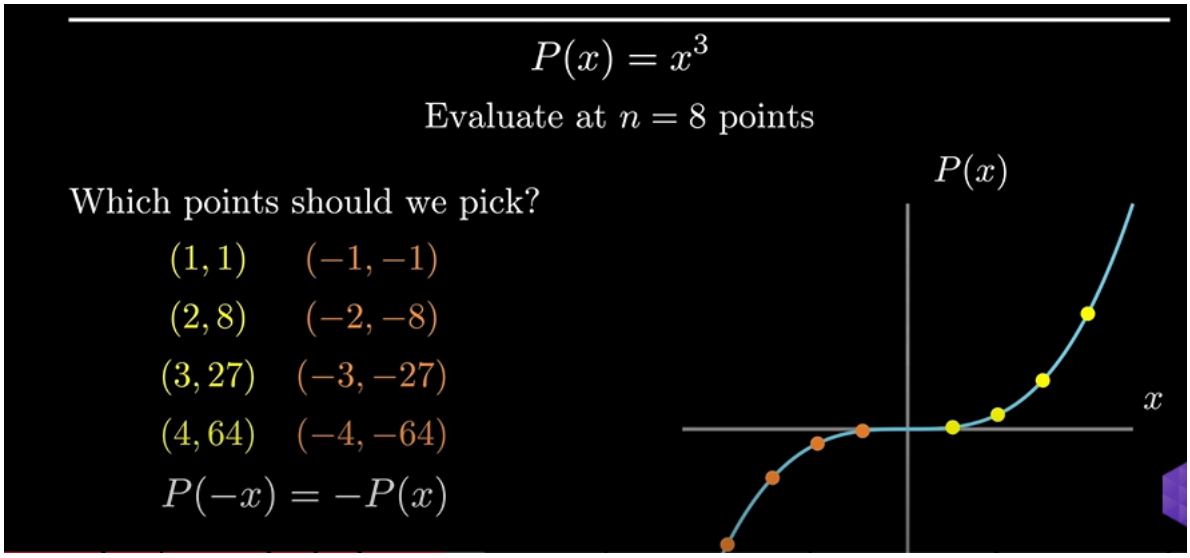


So as naïve approach has time complexity of O(d^2)
Where d is degree of polynomial.
Can we do better than O(d^2)

So let us take a even function



Now let us take odd polynomial



Now let us take a general example

$$P(x) = 3x^5 + 2x^4 + x^3 + 7x^2 + 5x + 1$$

Evaluate at n points $\pm x_1, \pm x_2, \ldots, \pm x_{n/2}$

$$P(x) = \underbrace{(2x^4 + 7x^2 + 1)}_{P_e(x^2)} + x\underbrace{(3x^4 + x^2 + 5)}_{P_o(x^2)}$$

$$P(x) = P_e(x^2) + xP_o(x^2)$$

$$\left.\begin{aligned} P(x_i) &= P_e(x_i^2) + x_i P_o(x_i^2) \\ P(-x_i) &= P_e(x_i^2) - x_i P_o(x_i^2) \end{aligned}\right\} \text{Lot of overlap!}$$

$$P_e(x^2) = 2x^2 + 7x + 1 \quad P_o(x^2) = 3x^2 + x + 5$$

$P_e(x^2)$ and $P_o(x^2)$ have degree 2!

$$P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

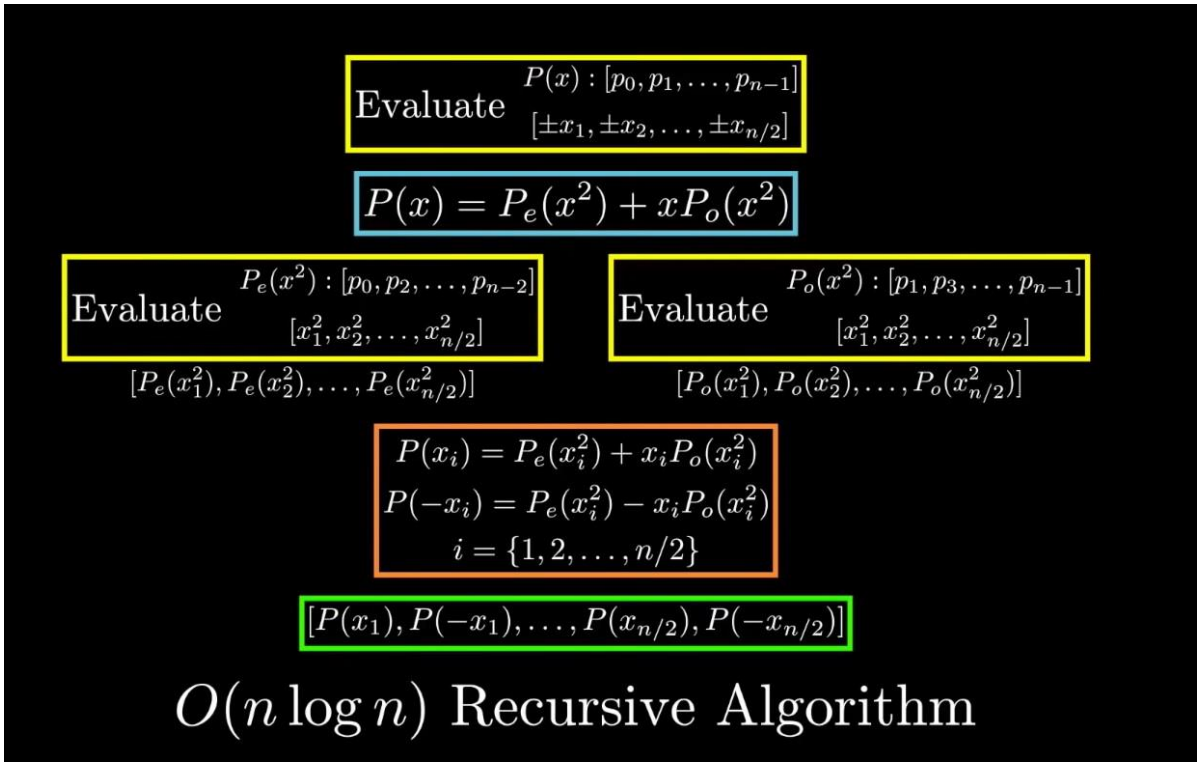
Evaluate at n points $\pm x_1, \pm x_2, \ldots, \pm x_{n/2}$

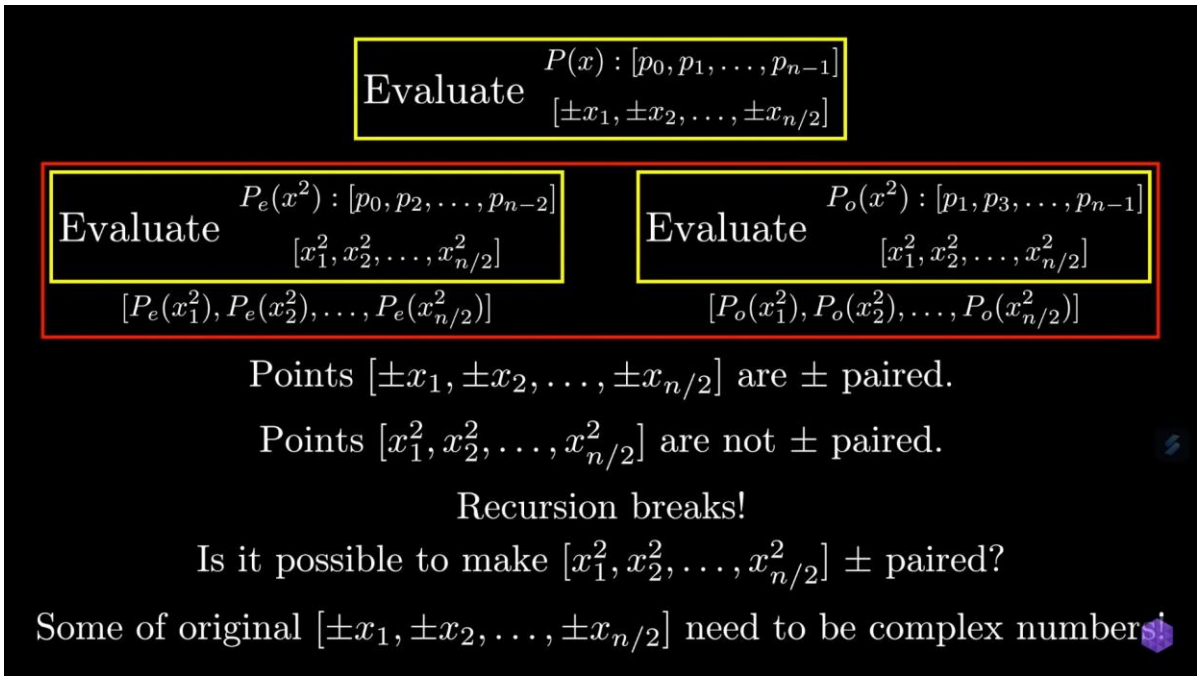
$$P(x) = P_e(x^2) + xP_o(x^2)$$

$$\left.\begin{aligned} P(x_i) &= P_e(x_i^2) + x_i P_o(x_i^2) \\ P(-x_i) &= P_e(x_i^2) - x_i P_o(x_i^2) \end{aligned}\right\} \text{Lot of overlap!}$$

$P_e(x^2)$ and $P_o(x^2)$ have degree $n/2 - 1$!

$$\underbrace{\text{Evaluate } P_e(x^2) \text{ and } P_o(x^2) \text{ each at } x_1^2, x_2^2, \ldots, x_{n/2}^2 \ (n/2 \text{ points})}_{\text{Same process on simpler problem}}$$





But which set of complex number should we choose?

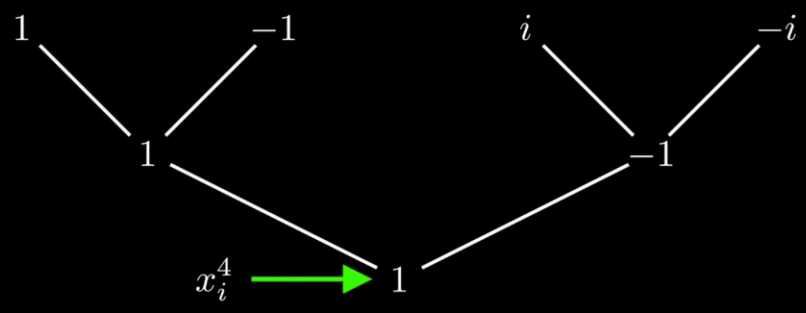
$P(x) = x^3 + x^2 - x - 1$

Alternative Perspective

Solution to $x^4 = 1$

Points are 4^{th} roots of unity!

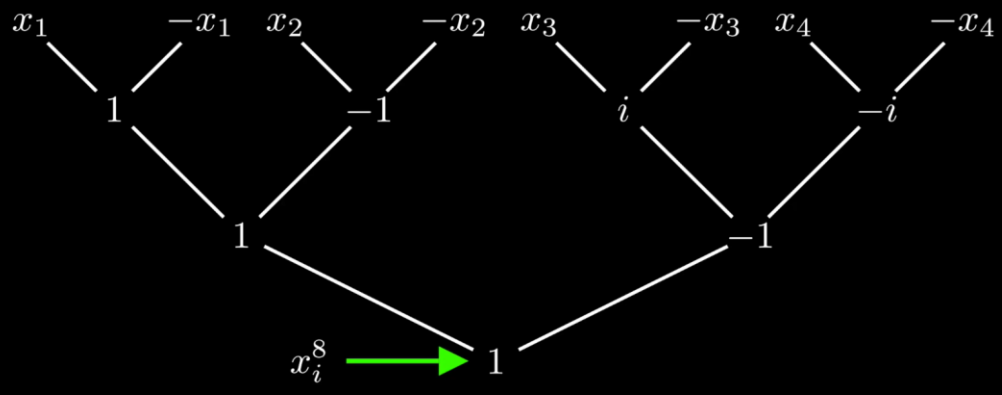
Does this generalize?

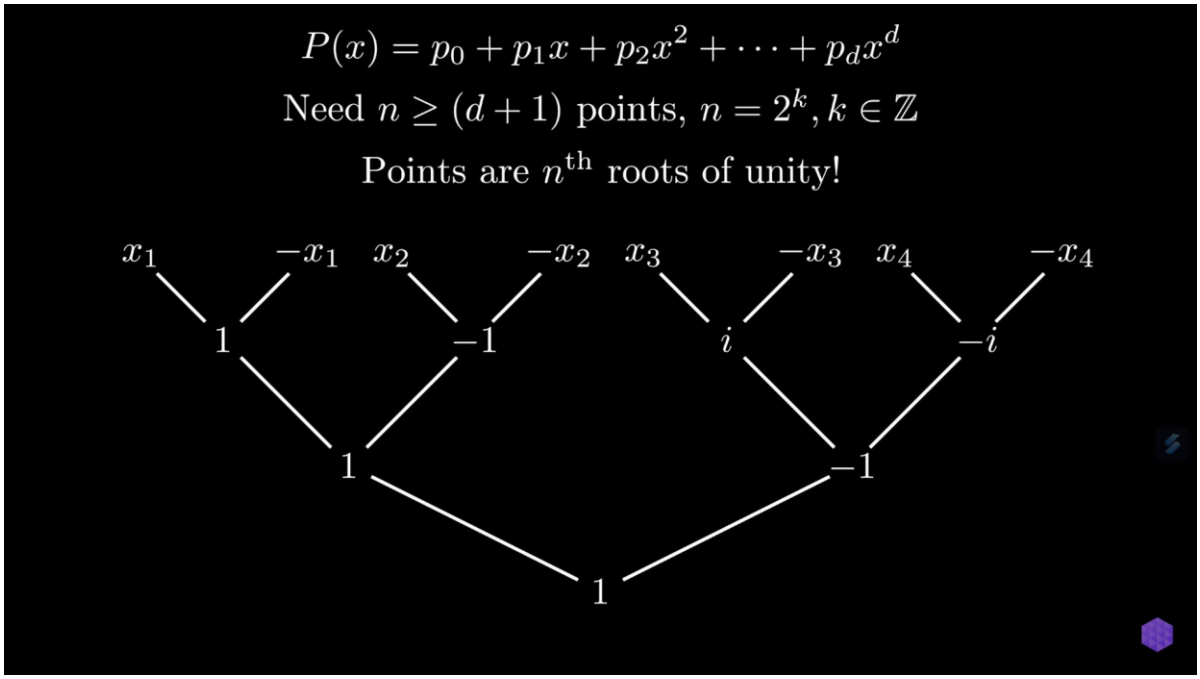


$P(x) = x^5 + 2x^4 - x^3 + x^2 + 1$

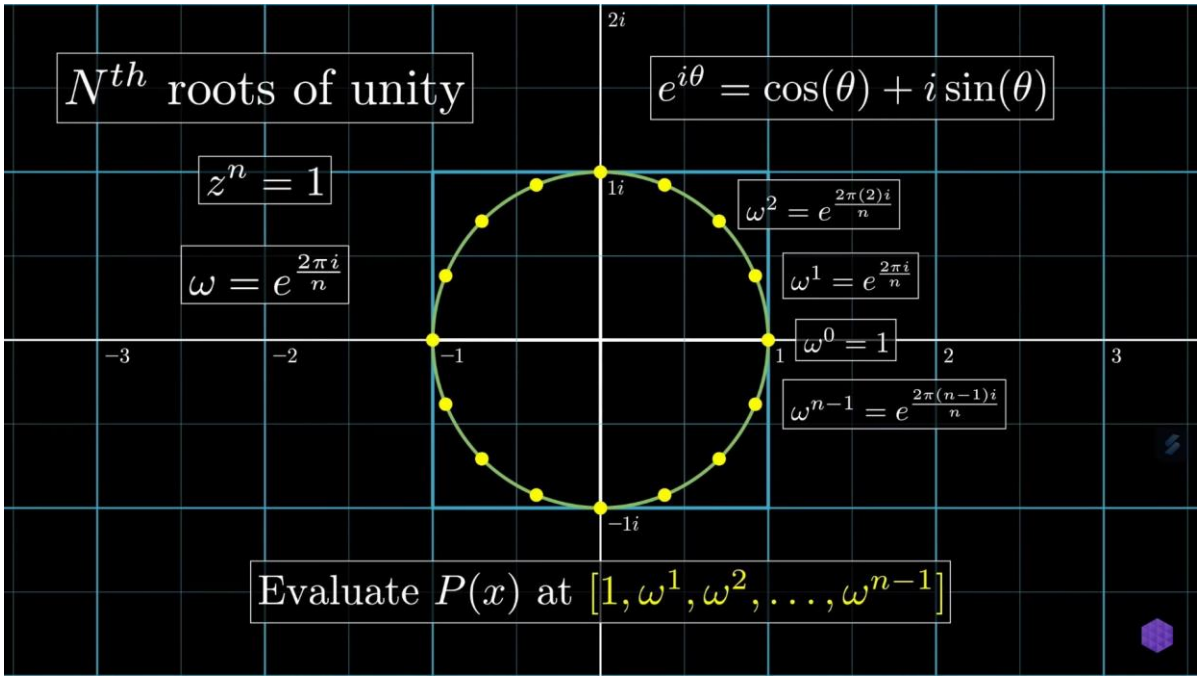
Need $n \geq 6$ points$\rightarrow$ let $n = 8$ (powers of 2 are convenient)

Points are 8^{th} roots of unity!



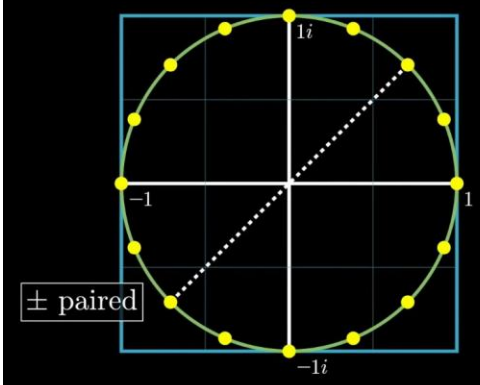


Nth root of Unity:

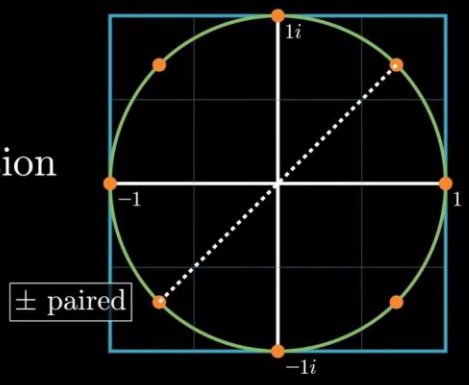


Why does this work?

$\omega^{j+n/2} = -\omega^j \rightarrow (\omega^j, \omega^{j+n/2})$ are $\pm$ paired



Recursion $\Longrightarrow$



Evaluate $P(x)$ at $[1, \omega^1, \omega^2, \ldots, \omega^{n-1}]$

n roots of unity

Evaluate $P_e(x^2)$ and $P_o(x^2)$ at $[1, \omega^2, \omega^4, \ldots, \omega^{2(n/2-1)}]$

$(n/2)$ roots of unity

FFT $P(x) : [p_0, p_1, \ldots, p_{n-1}]$

$\omega = e^{\frac{2\pi i}{n}} : [\omega^0, \omega^1, \ldots, \omega^{n-1}]$

$n = 1 \Rightarrow P(1)$

FFT $P_e(x^2) : [p_0, p_2, \ldots, p_{n-2}]$ $[\omega^0, \omega^2, \ldots, \omega^{n-2}]$

$y_e = [P_e(\omega^0), P_e(\omega^2), \ldots, P_e(\omega^{n-2})]$

FFT $P_o(x^2) : [p_1, p_3, \ldots, p_{n-1}]$ $[\omega^0, \omega^2, \ldots, \omega^{n-2}]$

$y_o = [P_o(\omega^0), P_o(\omega^2), \ldots, P_o(\omega^{n-2})]$

$x_j = \omega^j$

$-\omega^j = \omega^{j+n/2}$

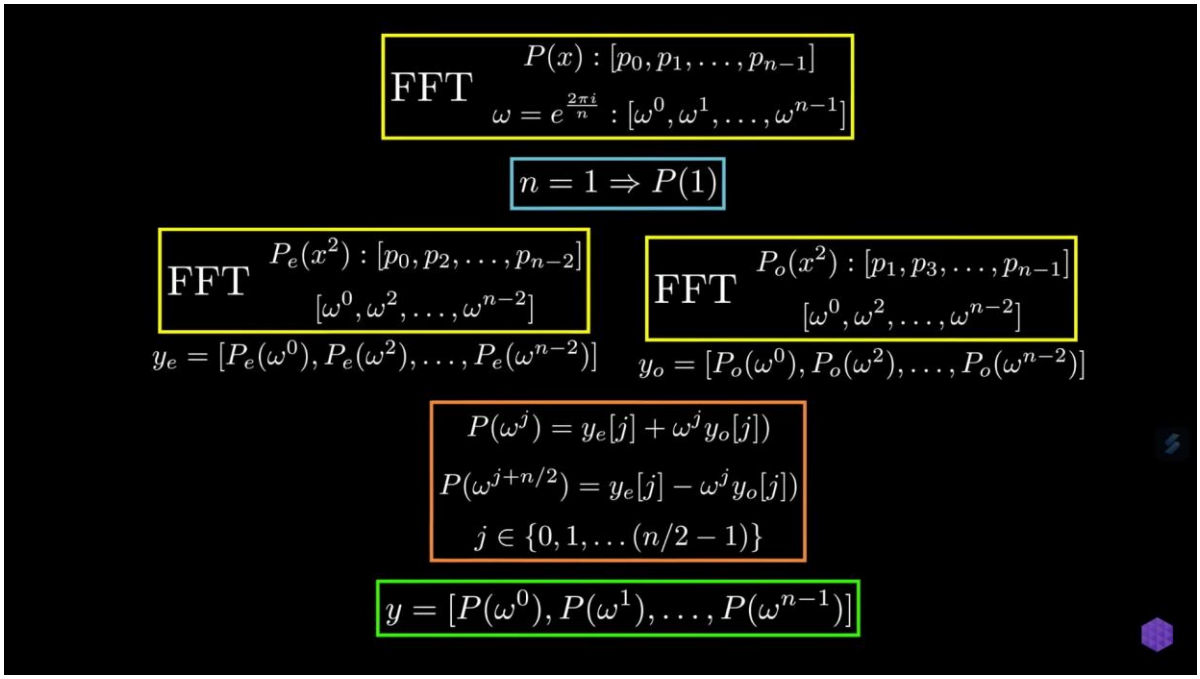
$$P(\omega^j) = P_e(\omega^{2j}) + \omega^j P_o(\omega^{2j})$$

$$P(\omega^{j+n/2}) = P_e(\omega^{2j}) - \omega^j P_o(\omega^{2j})$$

$$j \in \{0, 1, \ldots (n/2 - 1)\}$$

$y_e[j] = P_e(\omega^{2j})$

$y_o[j] = P_o(\omega^{2j})$



Step3:- **Point form to Multiplied Point form:**

Let us take a example:

Polynomial p1=x+3

Polynomial p2=2x+4

Multiplied polynomial point form p1*p2=(2x^2)+10x+12

Let us take represent p1,p2 in point form:

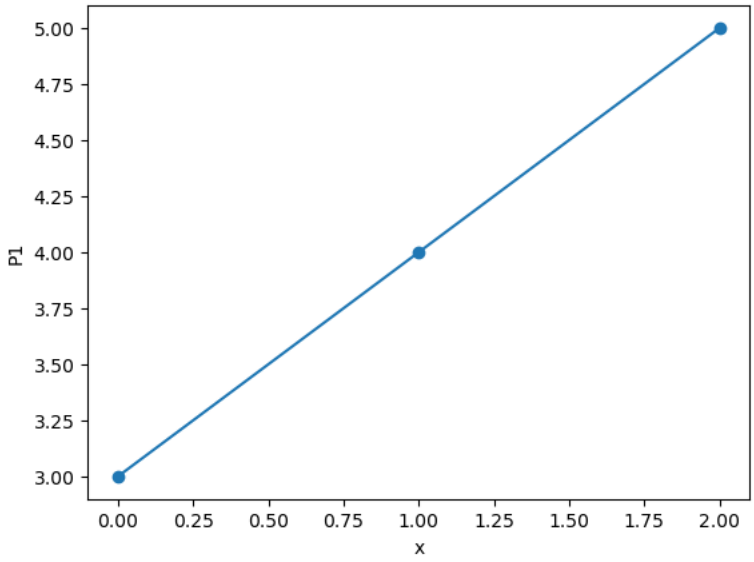
P1: (0,3), (1,4), (2,5)

P2: (0,4), (1,6), (2,8)

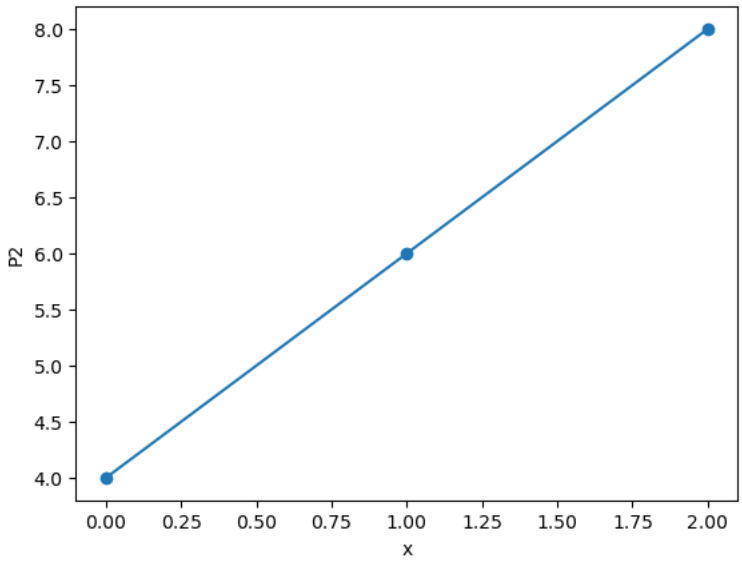
Multiplied polynomial point form:

P1*P2: (0,12), (1,24), (2,40)

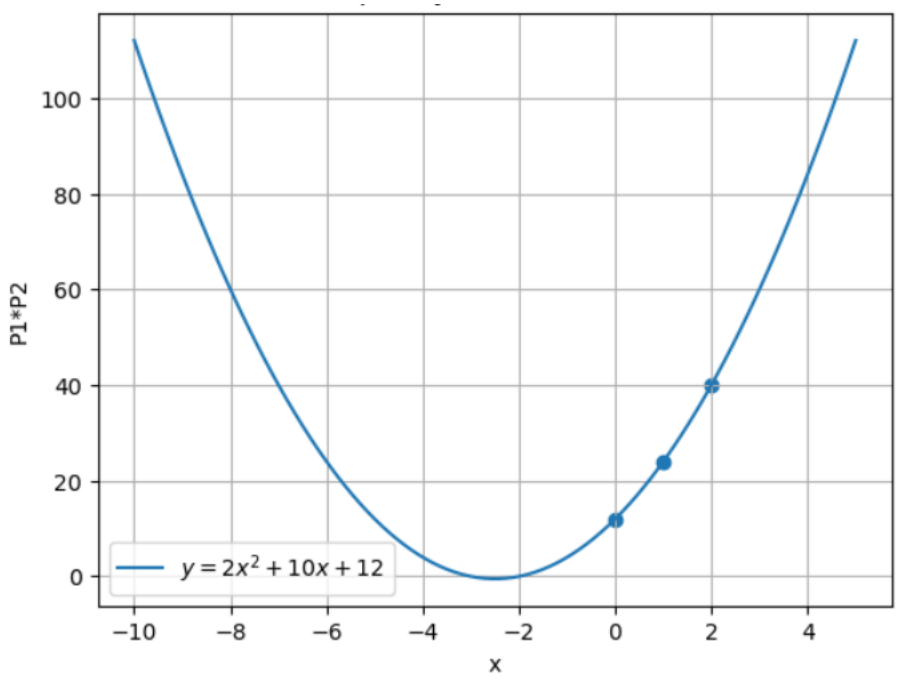
Graph for P1:



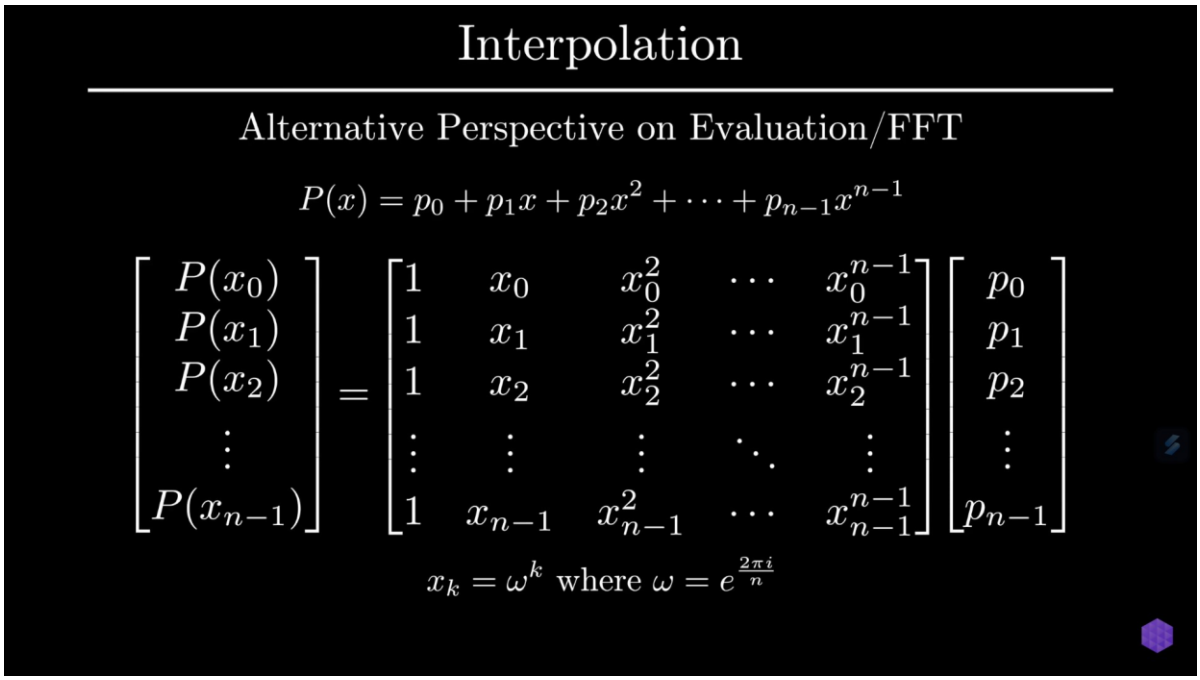
Graph for P2:

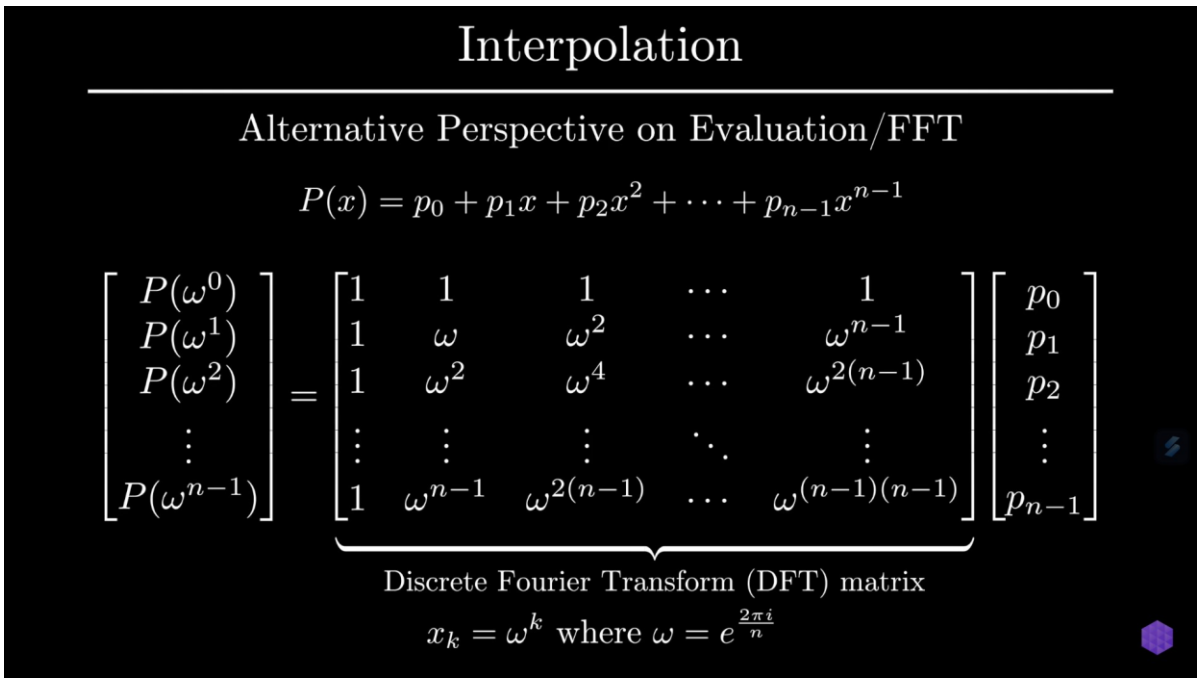


Graph for P1*P2:



Step3:- **Multiplied point form to Polynomial form Using IFFT:**





Interpolation

Interpolation involves inversing the DFT matrix

$$P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}^{-1} \begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix}$$

$$x_k = \omega^k \text{ where } \omega = e^{\frac{2\pi i}{n}}$$

$$x_k = \omega^k \text{ where } \omega = e^{\frac{2\pi i}{n}}$$

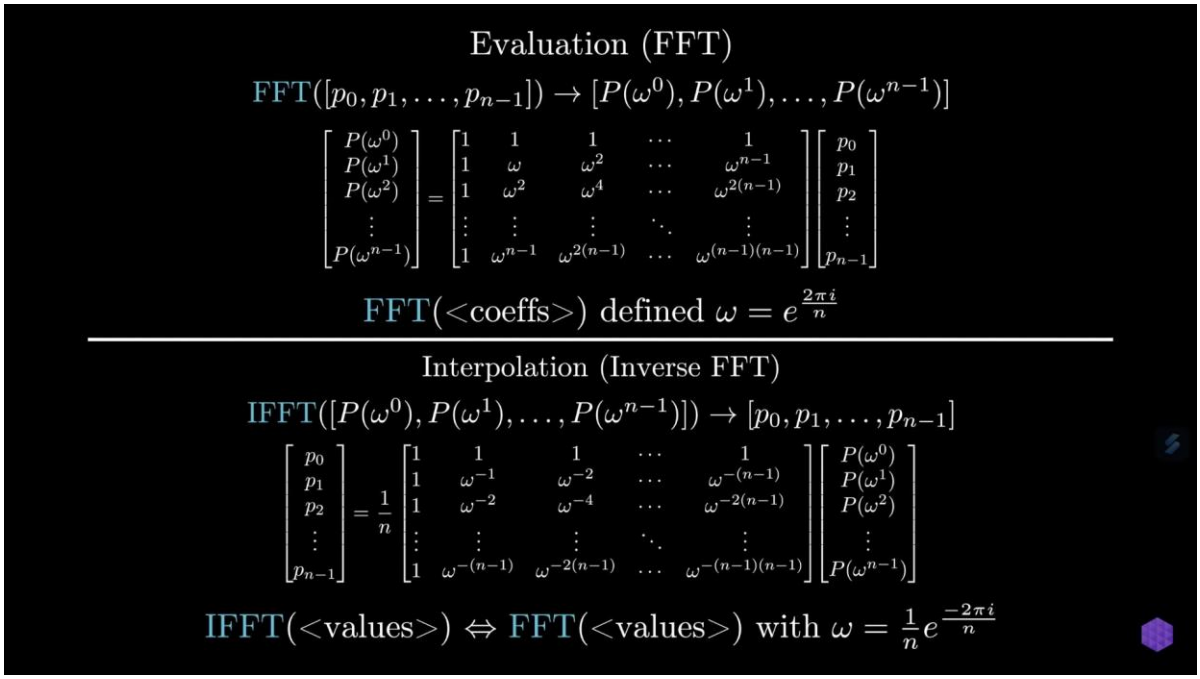
$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}^{-1} \begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix}$$

The inverse matrix and original matrix look quite similar!

Every ω in original matrix is now $\frac{1}{n}\omega^{-1}$



Now let us see code:

```
public class FFT
{
    public static class Complex
    {
        private final double re; // real part
        private final double im; // imaginary part


        public Complex(double real, double imag) {
            re = real;
            im = imag;
        }

        public String toString() {
            if (im == 0)
                return re + "";
            if (re == 0)
                return im + "i";
            if (im < 0)
                return re + " - " + (-im) + "i";
            return re + " + " + im + "i";
        }
        public Complex times(double alpha) {
            return new Complex(alpha * re, alpha * im);
        }

        public Complex conjugate() {
            return new Complex(re, -im);
        }

        public double abs() {
            return Math.sqrt((Math.pow(re,2)+Math.pow(im,2)));
        }

        public double phase() {
            return Math.atan2(im, re);
        }

        public Complex plus(Complex b) {
            Complex a = this;
            double real = a.re + b.re;
            double imag = a.im + b.im;
            return new Complex(real, imag);
        }

        public Complex minus(Complex b) {
            Complex a = this;
            double real = a.re - b.re;
            double imag = a.im - b.im;
            return new Complex(real, imag);
        }

        public Complex times(Complex b) {
            Complex a = this;
            double real = a.re * b.re - a.im * b.im;
            double imag = a.re * b.im + a.im * b.re;
            return new Complex(real, imag);
        }
```

```
        public Complex reciprocal() {
            double scale = re * re + im * im;
            return new Complex(re / scale, -im / scale);
        }

        public Complex sin() {
            return new Complex(Math.sin(re) * Math.cosh(im), Math.cos(re)
                    * Math.sinh(im));
        }


        public Complex cos() {
            return new Complex(Math.cos(re) * Math.cosh(im), -Math.sin(re)
                    * Math.sinh(im));
        }


        public Complex tan() {
            return sin().divides(cos());
        }
        public Complex divides(Complex b) {
            Complex a = this;
            return a.times(b.reciprocal());
        }


        public Complex exp() {
            return new Complex(Math.exp(re) * Math.cos(im), Math.exp(re)
                    * Math.sin(im));
        }

        public static Complex[] fft(Complex[] x) {
            int N = x.length;


            if (N == 1)
                return new Complex[] { x[0] };


            if (N % 2 != 0) throw new RuntimeException("N is not a power
of 2");
            // fft of even terms
            Complex[] even = new Complex[N / 2];
            for (int k = 0; k < N / 2; k++) {
                even[k] = x[2 * k];
            }
            Complex[] q = fft(even);

            // fft of odd terms
            Complex[] odd = even;
            for (int k = 0; k < N / 2; k++) {
                odd[k] = x[2 * k + 1];
            }
            Complex[] r = fft(odd);


            Complex[] y = new Complex[N];
```

```
            for (int k = 0; k < N / 2; k++) {
                double kth = -2 * k * Math.PI / N;
                Complex wk = new Complex(Math.cos(kth), Math.sin(kth));
                y[k] = q[k].plus(wk.times(r[k]));
                y[k + N / 2] = q[k].minus(wk.times(r[k]));
            }
            return y;
        }


        public static Complex[] ifft(Complex[] x) {
            int N = x.length;
            Complex[] y = new Complex[N];


            for (int i = 0; i < N; i++) {
                y[i] = x[i].conjugate();
            }


            y = fft(y);


            for (int i = 0; i < N; i++) {
                y[i] = y[i].conjugate();
            }


            for (int i = 0; i < N; i++) {
                y[i] = y[i].times(1.0 / N);
            }

            return y;

        }


        public static void print(Complex[] arr, String s) {
            System.out.println(s);
            for (int i = 0; i < arr.length; i++)
System.out.println(arr[i]);
            System.out.println();
        }

        public static void main(String[] args) {
            int N = 4;
            Complex[] p1 = new Complex[N];
            Complex[] p2 = new Complex[N];
            int[] arr1=new int[]{1,1,0,0};
            int[] arr2=new int[]{1,1,0,0};

            // original data
            System.out.println("Given Polynomial");
            for (int i = 0; i < N; i++) p1[i] = new Complex(arr1[i], 0);
            for (int i = 0; i < N; i++) p2[i] = new Complex(arr2[i], 0);
            print(p1, "p1");
            print(p2, "p2");
```

```
            //******STEP:1 : Polynomial to point form
            System.out.println("Fourier Transform of Given Polynomial");
            Complex[] y1 = fft(p1);
            print(y1, "y1 = fft(p1(x)");
            Complex[] y2 = fft(p2);
            print(y2, "y2 = fft(p2(x)");




            //******STEP:2 : Point form Multiplication
            System.out.println("Point multiplication form of given
polynomial");
            Complex[] y = new Complex[N];
            for (int i = 0; i < N; i++)  y[i] = y1[i].times(y2[i]);

            print(y, "multiplied point form");



           //******STEP:3 : Taking the Inverse
            System.out.println("Inverse Fourier Transform of multiplied
point form");
            Complex[] p = ifft(y);
            print(p, "p = ifft(y)");
            System.out.println("This is multiplied ploynomial");

        }

    }
}
```

AKS primality test

In this test we are using a concept :
If n is prime:
(x – 1)^n – (x^n – 1) is divisible by n.

C(n,1) x^(n-1) + C(n,2)x^(n-2) -….+(-1)^(n-1)C(n,n-1)x

It means all the coefficients in this expansion must be divisible by n:
So C(n,1), C(n,2), C(n,3) …….C(n,n-1), C(n,n) all of these terms must be divisible by n.

So this algorithm will have O(n^2) complexity.

Time complexity: $O(n^2)$
Auxiliary space: O(1) as space taken is constant.

```java
import java.util.ArrayList;

public class AKS_Test {
    public static void main(String[] args) {
        System.out.println("All primes smaller "
                + "than 20: ");
        ArrayList<Integer> list=new ArrayList<>();
        for (int n = 1; n < 20; n++) {
            if (solve(n)) {
                list.add(n);
            }
        }
        System.out.println(list);
        System.out.println(list.size());
    }
    static boolean solve(int n) {
        if(n==0||n==1) return false;
        for(int i=1;i<=n/2;i++) {
            if(!is(n,i)) return false;
        }
        return true;
    }
    //function to check weather n divides it's coefficient
    static boolean is(int n,int r) {
        long a=fact(n)/(fact(r)*fact(n-r));
        if(a%n==0) return true;
        return false;
    }
    //function to claculate factorial
    static long fact(int n) {
        long ans=(long)1;
        for(int i=2;i<=n;i++) {
            ans*=i;
        }
        return ans;
    }
}
```

Problems:

P1: Describe an algorithm that determines whether a given set X of n integers contains two elements whose sum is zero, in O(n log n) time.

```java
import java.util.*;
public class Exam1_a {
    public static void main(String[] args) {
        int[] arr=new int[]{2,45,4,67,21,12,102};
        System.out.println(func1(arr,5));
    }
    static boolean func1(int[] arr,int k) {
        int n=arr.length;
        Arrays.sort(arr);
        for(int i=0;i<n;i++) {
            if(bs(arr,i+1,n-1,k-arr[i])) return true;
        }
        return false;
    }
    static boolean bs(int[] arr,int s,int e,int pivot) {
        while(s<=e) {
            int m=s+(e-s)/2;
            if(arr[m]>pivot) e=m-1;
            else if(arr[m]<pivot) s=m+1;
            else return true;
        }
        return false;
    }
}
```

P2. Describe an algorithm that determines whether a given set X of n integers contains three elements whose sum is zero, in O(n2) time.

```java
import java.util.*;
public class Exam1_b {
    public static void main(String[] args) {
        int[] arr=new int[]{1,2,2,3,4};
        System.out.println(func1(arr,5));
    }
    static boolean func1(int[] arr,int k) {
        Arrays.sort(arr);
        int n=arr.length;
        for(int i=0;i<n;i++) {
            int pivot=k-arr[i];
            if(is(arr,pivot,i+1,n-1)) return true;
        }
        return false;
    }
    static boolean is(int[] arr,int pivot,int s,int e) {
        while(s<e) {
            int sum=arr[s]+arr[e];
            if(sum>pivot) e--;
            else if(sum<pivot) s++;
            else return true;
        }
        return false;
    }
}
```

P3: Assume that the input set X of n integers contains only integers between −5000n and 6000n. Describe an algorithm that determines whether X contains three elements whose sum is zero, in O(n log n) time.

Solution :-Now if we see 3 sum problem, naïve approach take O(n^3).
If we optimize then it can be done in O(n*n(logn))
If we optimize it more then it can be done in O(n*n)

Now if we try to optimize it further then it is not possible using normal algorithms.
But If we use FFT then it can be done in O(nlogn)

How it is possible?
Let us try to understand with steps with example array Array=[1,-1,0,-2]:-

Step1:-Sort the array Time Complexity:- O(nlogn)

After sorting Array=[-2,-1,0,1]

Step2:-Now just declare 3 arrays A, B, C where the power of x comes from element of given array. Time Complexity:- O(n)

A=[x^-2, x^-1, x^0, x^1]
B=[x^-2, x^-1, x^0, x^1]
C=[x^-2, x^-1, x^0, x^1]

Step3:- Now make a array for storing polynomials coefficients
Time Complexity:- O(n)
Arr1=[1,1,0,1]
Arr2=[1,1,0,1]
Arr3=[1,1,0,1]

Step4:- Mutiply arr1 with arr2 using fft but donot multiply elements at same I Index. Let res1=Arr1*Arr2 res is array storing coefficients of Arr1*Arr2 And res2=Arr3*res1, where res2 store coefficients of arr1*arr2*arr3 and res store A*B*C Time Complexity:- O(nlogn)

Res=A*B*C=[6*x^(-3)+ 6*x^(-2)+ 8*x^(-1)+2]

Step4:- Just check that there is a constant term in result if present then Return true else return false. Time Complexity:- O(n)

As res contain constant term so return yes.

So overall **Time Complexity: O(NlogN)**

P4:- Suppose we are given a bit string B[1, . . . , n]. A triple of distinct indices 1 ≤ i < j < k ≤ n is called a well-spaced triple in B if B[i] = B[j] = B[k] = 1 and k − j = j − i.

Example: Let n = 10 and B[1, . . . , 10] = 0100110011, where B[1] = 0, B[2] = 1, . . . , B[9] =1, B[10] = 1. Since B[2] = B[6] = B[10] = 1 and 10 − 6 = 6 − 2 = 4, we can say that (2, 6, 10) isa well-spaced triple.

A. **Describe a brute-force algorithm to determine whether B has a well-spaced triple in O(n2) time.**

```
    public class Exam3_a {
    public static void main(String[] args) {
        int[] arr=new int[]{1,0,1,1,1,0,1};
        System.out.println(func1(arr));
    }
    static int func1(int[] arr) {
        int n=arr.length;
        int c=0;
        for(int i=0;i<n-2;i++) {
            if(arr[i]==1) {
                for(int k=i+2;k<n;k++) {
                    if(arr[k]==1&&((i+k)&1)==0) {
                        int j=(i+k)/2;
                        if(arr[j]==1) c++;
                    }
                }
            }
        }
        return c;
    }
}
```

(b) Describe an algorithm to determine whether B has a well-spaced triple in O(n log n) time. Also, determine the number of well-spaced triples in B in O(n log n) time.

NP Completeness

Now In computer science as some problems are solved and some are unsolved.

Solved means if they were solvable then they are proved mathematically and there algorithms are found and those problem which cannot be solved are proved mathematically that these cannot be solved.
And remaining problem are unsolved.

Now as for each algorithm there is some time complexity. Time complexity can be polynomial or non polynomial.

Polynomial Time (P):
Algorithm which have polynomial complexity.
So polynomial time algorithm are algorithm which has Time complexity of **O(n^k)**

Example of Algorithm with polynomial time complexity:-
Linear search O(N)
Binary search O(logN)
Merge Sort (NlogN)
Minimum spanning tree O(ElogE)
Matrix multiplication O(N^3)
Bubble sort(N^2)

Non Polynomial Time (NP):
Algorithm that cannot be solved in polynomial time or can be solved in exponential time.
Exponential time complexity: O(a^n)

Example:
Sum of subsets: O(2^n)
Hamiltonian cycle: O(2^n)
Graph Coloring: O(2^n)

Now with this discussion term Deterministic and non-deterministic term come into picture.

Deterministic:
Deterministic is a algorithm in which every step is determined.
For example : binary search

```
S=0, e=arr.length-1;
While(s<=e) {
    m=s+(e-s)/2;
    if(arr[m]==target) return true;
    else if(arr[m]<target) s=m+1;
    else e=m-1;
}
return false;
```

Now this is a deterministic algorithm as every step can be determined.

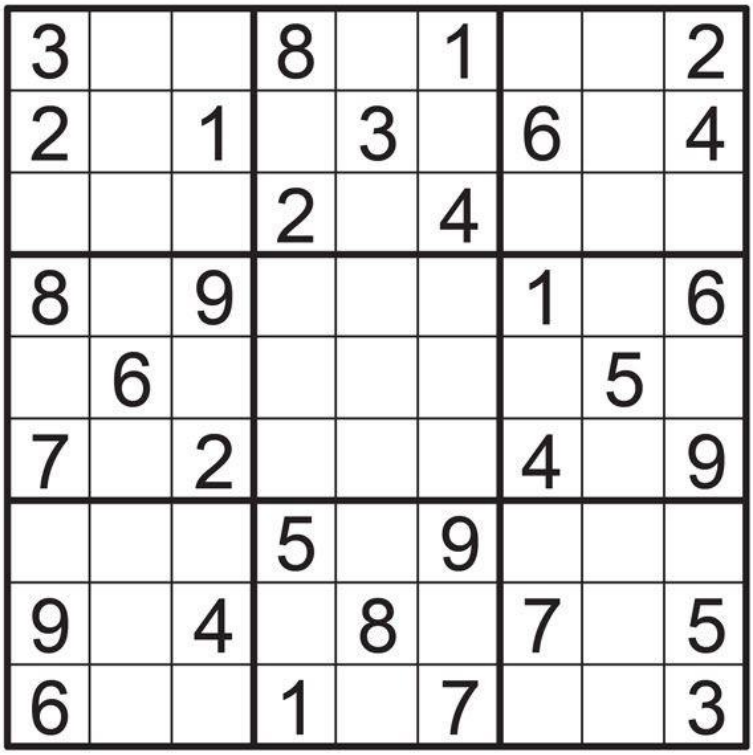
Non-Deterministic:
Non-Deterministic is a algorithm in which there is one or more than one step that are undetermined.
For example : searching a element in array in O(1) complexity

```
Algorithm Nsearch(A,n,key) {
        J=choice();--------nondeterministic step
       if(key==A[j]) {
            write(j);
             success();
       }
       write(0);
       Failure();
}
```

Here every step has O(1) complexity.
Now this is a non-deterministic algorithm as 1 step are undetermined.

Now why we require Non-deterministic algorithms.
Now if researchers try to get searching algorithm in O(1) complexity. Then they have to work on only the 1 undeterministic step if they solve it then searching algorithm can be possible with O(1) complexity.

Non deterministic Polynomial time:-
These are the problem which cannot be solved in polynomial time but can be verified in polynomial time.
Eg:- sudoku problem



So here as if we try to solve this problem than it will take exponential time. But after solving this problem verification takes polynomial time.

Now before moving further to understand np complete let us discuss Reduction.

Reduction:-

Now as we know that there are many problem in np category. But there can be problems that are interrelated to each other. It means if one can be solved then another can also be solved let us take a basic example.

There is a satisfiability problem. Suppose we try to find is there a way to assign values to variable of function such that it evaluates true. Let us take of 3 vaiables x, y, z.

Example: $f(x, y, z) = (x \vee (y \wedge z)) \wedge (x \wedge z)$

X	y	z	$x \vee (y \wedge z)$	$x \wedge z$	f(x, y, z)
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	0	0
1	1	1	1	1	1

Now here ,V-> OR operator, ^ ->AND operator

Here as we see can see that it has polynomial time complexiy.

Now there is a 0/1 knapsack problem:
Maximum weight is given with each weight associated and profit with it. So we are just trying to find maximum profit.

So this can be done using the fact that we are taking a bag or not. Taking means 1 or not taking means 0.

So this problem is reducible to SAT problem if SAT can be solved this can also be solved.

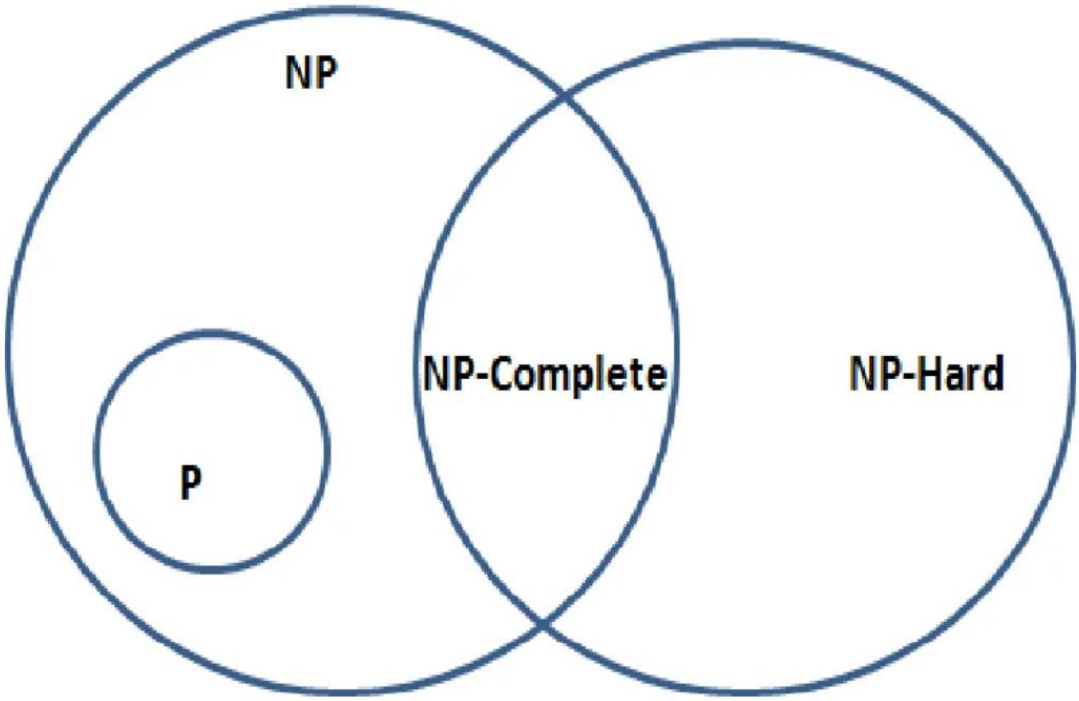
After seeing above example we have understand what reduction is but we to take care of one thing that all of the stuff we are doing here to solve a problem in polynomial time but if reduction takes exponential time then it doesnot help. So reduction step must be in polynomial time.

NP-hard:-
A problem is said to np-hard if every problem in NP can be polynomially reduced to it.

NP-Complete:

Now as there are some problems which have exponential time complexity and researchers make a non-deterministic algorithm for a problem then it come under np-complete category.
Now all the problem that are reducible to this come under np-complete category. These are problems that are np-hard and np both.

Let us see a diagram clearly explaining about all these terms.


NP
NP-Complete
NP-Hard
P