```
static void test_poly_algebra()
        {
            Polynomial A = new Polynomial(4, -3, 2);
            Polynomial B = new Polynomial(4, -3, 2, -1);
            Polynomial P = A * B;
            Polynomial Q = P / A;
            bool correct = Q == B;

            A = FromRoots(2,-1,1,1,1,0,0);
            B = 4*FromRoots(-0.5,0.5);
            P = A * B;
            Q = (Polynomial) (P / B);
            correct = Q == A;

            var dP = Derivative(P);
            var R = Integral(dP)+P[P.Degree];
            correct = P == R;

        }
```

```
using System;

namespace MMP.Tools
{
    public static class General
    {
        public static void Swap<T>(ref T a, ref T b)
```

```
        {
            T tmp;
            tmp = a;
            a = b;
            b = tmp;
        }

        public static string Print<T>(T a, int p, int n) where T : IComparable<T>
        {
            string s = a.ToString().Replace("-", "");

            if (p > 0 && s == "1" ) s = "";
            if (p < n)
                if (a.CompareTo(default(T)) < 0)
                    s = "-" + s;
                else
                    s = "+" + s;
            if (p > 1)
                return s + string.Format($"x^{p}");
            else if (p == 1)
                return s + "x";
            else
                return s;
        }

    }
}



using System;
```

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static MMP.Tools.General;

namespace MMP.Tools
{
    /// <summary>
    /// Generic Polynomial class
    /// a[0]*x^n+a[1]*x^(n-1)+...+a[n];
    /// </summary>
    /// <typeparam name="Ta">Type of coeeficients</typeparam>
    /// <typeparam name="Tx">Type of arguments</typeparam>
    [DebuggerDisplay("{ToString(), nq}")]
    public class Polynomial<Ta, Tx> : List<Ta>, IEquatable<Polynomial<Ta, Tx>>
        where Ta : struct, IEquatable<Ta>, IComparable<Ta>
    {
        public static Ta Zero { get { return default(Ta); } }

        public Polynomial(bool empty = false)
        {
            if (!empty)
                Add(Zero);
        }

        public Polynomial(params Ta[] coefficients)
        {
            AddRange(coefficients);
            Trim();
```

```
}

/// <summary>
/// Copy constructor
/// </summary>
/// <param name="other"></param>
public Polynomial(Polynomial<Ta, Tx> other)
{
    AddRange(other);
}

public bool Trim()
{
    int count = FindIndex(a => !a.Equals(Zero));
    bool b = count > 0;
    if (b)
        RemoveRange(0, count);
    if (count == -1)
    {
        Clear();
        Add(Zero);
    }
    return b;
}

public bool Equals(Polynomial<Ta, Tx> other)
{
    if (Degree != other.Degree)
        return false;
    for (int i = 0; i < Count; i++)
    {
```

```
            if (!this[i].Equals(other[i]))
                return false;
        }
        return true;
    }

    public override bool Equals(object obj)
    {
        return base.Equals(obj);
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }

    public static bool operator ==(Polynomial<Ta, Tx> p, Polynomial<Ta, Tx> q)
    {
        return p.Equals(q);
    }

    public static bool operator !=(Polynomial<Ta, Tx> p, Polynomial<Ta, Tx> q)
    {
        return !p.Equals(q);
    }

    public int Degree
    {
        get
        {
            if (Count > 1)
```

```
                    return Count - 1;
                else if (this[0].Equals(Zero))
                    return -1;
                else
                    return 0;
            }
        }

        public override string ToString()
        {
            int n = Degree;
            if (n == -1)
                return "0";
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i<Count; i++)
                if(!this[i].Equals(Zero))
                    sb.Append(Print(this[i], n-i, n));
            return sb.ToString();
        }

    }

}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MMP.Tools;
```

```
using static MMP.Tools.General;

namespace MMP.Tools.DoubleDouble
{
    using Ta = System.Double;
    using Tx = System.Double;
    public class Polynomial : Polynomial<Ta, Tx>
    {
        Polynomial(bool empty = false) : base(empty) { }
        public Polynomial(params Ta[] coefficients) : base(coefficients) { }
        public Polynomial(Polynomial other) : base(other) { }

        /// <summary>
        /// Horner's evaluation
        /// </summary>
        /// <param name="x"></param>
        /// <returns></returns>
        public Tx Eval(Tx x)
        {
            Tx r = this[0];
            for (int i = 1; i < Count; i++)
            {
                r *= x;
                r += this[i];
            }
            return r;
        }

        public static Polynomial operator +(Polynomial p, Polynomial q)
        {
            if (p.Degree <= q.Degree)
```

```
            Swap(ref p, ref q);
        var r = new Polynomial(p);
        int ins = r.Degree - q.Degree;
        foreach (var item in q)
            r[ins++] += item;
        r.Trim();
        return r;
    }

    public static Polynomial operator -(Polynomial p, Polynomial q)
    {
        return p + (-q);
    }

    public static Polynomial operator +(Polynomial p, Ta a)
    {
        Polynomial q = new Polynomial(p);
        q[q.Degree] += a;
        return q;
    }

    public static Polynomial operator +(Ta a, Polynomial p)
    {
        return p + a;
    }

    public static Polynomial operator *(Polynomial p, Polynomial q)
    {
        int n = p.Degree + q.Degree + 1;
        double[] r = new double[n];
        for (int k = 0; k < n; k++)
```

```
        for (int i = 0; i < k + 1; i++)
            if (i <= p.Degree && k - i <= q.Degree)
                r[k] += p[i] * q[k - i];
    return new Polynomial(r);
}

public static Polynomial Taylor(params double[] derivatives)
{
    long fac = 1;
    for (int i = 2; i < derivatives.Length; i++)
    {
        fac = fac * i;
        derivatives[i] /= fac;
    }
    return new Polynomial(derivatives.Reverse().ToArray());
}

public static Polynomial Sin(int degree)
{
    double[] derivatives = new double[degree + 1];
    for (int i = 1; i <= degree; i += 4)
        derivatives[i] = 1;
    for (int i = 3; i <= degree; i += 4)
        derivatives[i] = -1;
    return Taylor(derivatives);
}

public static Polynomial operator *(Polynomial p, Ta q)
{
    Polynomial r = new Polynomial(true);
    foreach (var item in p)
```

```
    {
        r.Add(item * q);
    }
    return r;
}
public static Polynomial operator *(Ta q, Polynomial p)
{
    return p * q;
}

public static Polynomial operator /(Polynomial p, Ta q)
{
    if (q == 0f)
        throw new DivideByZeroException();
    return 1 / q * p;
}

public static RatioFunc operator /(Polynomial p, Polynomial q)
{
    Polynomial p1;
    int dpq = p.Degree - q.Degree;
    if (dpq > -1)
        p1 = new Polynomial(true);
    else
        p1 = new Polynomial();
    while (dpq > -1)
    {
        var c = p[0] / q[0];
        p1.Add(c);
        var tmp = c * q;
        for (int i = 0; i < dpq; i++) tmp.Add(Zero);
```

```
            p = p - tmp;
            int pq1 = p.Degree - q.Degree;
            int def = dpq - pq1;
            if (pq1 > -1)
                for (int i = 0; i < def - 1; i++) p1.Add(0);
            else
                for (int i = 0; i < dpq; i++) p1.Add(0);
            dpq = pq1;

        }
        return new RatioFunc { p = p1, r = p, q = q };
    }

    public static Polynomial operator -(Polynomial p)
    {
        return -1f * p;
    }

    public static Polynomial Derivative(Polynomial p)
    {
        Polynomial dp = new Polynomial(true);
        int deg = p.Degree;
        for (int i = 0; i < deg; i++)
            dp.Add((deg - i) * p[i]);
        return dp;
    }

    public static Polynomial Integral(Polynomial p)
    {
        Polynomial P = new Polynomial(true);
        int len = p.Count;
```

```
            for (int i = 0; i < len; i++)
                P.Add(p[i] / (len - i));
            P.Add(0f);
            return P;
        }

        public static Polynomial Monom(int degree)
        {
            Polynomial p = new Polynomial();
            p[0] = 1f;
            for (int i = 0; i < degree; i++)
                p.Add(0f);
            return p;
        }


        public static Polynomial LinearFactor(Ta a)
        {
            Polynomial p = new Polynomial();
            p[0] = 1;
            p.Add(-a);
            return p;
        }

    }
}


using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Numerics;
using System.Text;
using System.Threading.Tasks;
using MMP.Tools;
using static MMP.Tools.General;

namespace MMP.Tools.DoubleComplex
{
    using Ta = System.Double;
    using Tx = Complex;
    public partial class Polynomial : Polynomial<Ta, Tx>
    {
        Polynomial(bool empty = false) : base(empty) { }
        public Polynomial(params Ta[] coefficients) : base(coefficients) { }
        public Polynomial(Polynomial other) : base(other) { }

        /// <summary>
        /// Horner's evaluation
        /// </summary>
        /// <param name="x"></param>
        /// <returns></returns>
        public Tx Eval(Tx x)
        {
            Tx r = this[0];
            for (int i = 1; i < Count; i++)
            {
                r *= x;
                r += this[i];
            }
            return r;
        }
```

```
public static Polynomial operator +(Polynomial p, Polynomial q)
{
    if (p.Degree <= q.Degree)
        Swap(ref p, ref q);
    var r = new Polynomial(p);
    int ins = r.Degree - q.Degree;
    foreach (var item in q)
        r[ins++] += item;
    r.Trim();
    return r;
}

public static Polynomial operator -(Polynomial p, Polynomial q)
{
    return p + (-q);
}

public static Polynomial operator +(Polynomial p, Ta a)
{
    Polynomial q = new Polynomial(p);
    q[q.Degree] += a;
    return q;
}

public static Polynomial operator +(Ta a, Polynomial p)
{
    return p + a;
}

public static Polynomial operator *(Polynomial p, Polynomial q)
```

```
{
    int n = p.Degree + q.Degree + 1;
    double[] r = new double[n];
    for (int k = 0; k < n; k++)
        for (int i = 0; i < k + 1; i++)
            if (i <= p.Degree && k - i <= q.Degree)
                r[k] += p[i] * q[k - i];
    return new Polynomial(r);
}

public static Polynomial Taylor(params double[] derivatives)
{
    long fac = 1;
    for (int i = 2; i < derivatives.Length; i++)
    {
        fac = fac * i;
        derivatives[i] /= fac;
    }
    return new Polynomial(derivatives.Reverse().ToArray());
}

public static Polynomial Sin(int degree)
{
    double[] derivatives = new double[degree + 1];
    for (int i = 1; i <= degree; i += 4)
        derivatives[i] = 1;
    for (int i = 3; i <= degree; i += 4)
        derivatives[i] = -1;
    return Taylor(derivatives);
}
```

```
public static Polynomial operator *(Polynomial p, Ta q)
{
    Polynomial r = new Polynomial(true);
    foreach (var item in p)
    {
        r.Add(item * q);
    }
    return r;
}
public static Polynomial operator *(Ta q, Polynomial p)
{
    return p * q;
}

public static Polynomial operator /(Polynomial p, Ta q)
{
    if (q == 0f)
        throw new DivideByZeroException();
    return 1 / q * p;
}

public static RatioFunc operator /(Polynomial p, Polynomial q)
{
    Polynomial p1;
    int dpq = p.Degree - q.Degree;
    if (dpq > -1)
        p1 = new Polynomial(true);
    else
        p1 = new Polynomial();
    while (dpq > -1)
    {
```

```
            var c = p[0] / q[0];
            p1.Add(c);
            var tmp = c * q;
            for (int i = 0; i < dpq; i++) tmp.Add(Zero);
            p = p - tmp;
            int pq1 = p.Degree - q.Degree;
            int def = dpq - pq1;
            if (pq1 > -1)
                for (int i = 0; i < def - 1; i++) p1.Add(0);
            else
                for (int i = 0; i < dpq; i++) p1.Add(0);
            dpq = pq1;

        }
        return new RatioFunc { p = p1, r = p, q = q };
    }

    public static Polynomial operator -(Polynomial p)
    {
        return -1f * p;
    }

    public static Polynomial Derivative(Polynomial p)
    {
        Polynomial dp = new Polynomial(true);
        int deg = p.Degree;
        for (int i = 0; i < deg; i++)
            dp.Add((deg - i) * p[i]);
        return dp;
    }
```

```
        public static Polynomial Integral(Polynomial p)
        {
            Polynomial P = new Polynomial(true);
            int len = p.Count;
            for (int i = 0; i < len; i++)
                P.Add(p[i] / (len - i));
            P.Add(0f);
            return P;
        }

        public static Polynomial Monom(int degree)
        {
            Polynomial p = new Polynomial();
            p[0] = 1f;
            for (int i = 0; i < degree; i++)
                p.Add(0f);
            return p;
        }


        public static Polynomial LinearFactor(Ta a)
        {
            Polynomial p = new Polynomial();
            p[0] = 1;
            p.Add(-a);
            return p;
        }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

namespace MMP.Tools.DoubleComplex
{
    using Ta = System.Double;
    using Tx = Complex;
    public partial class Polynomial : Polynomial<Ta, Tx>
    {
        public static Polynomial FromRoots(params Complex[] roots)
        {
            Polynomial P = new Polynomial(false);
            P.Add(1);
            foreach (var root in roots)
            {
                if (root.Imaginary == 0)
                    P *= LinearFactor(root.Real);
                else
                    P *= QuadraticFactor(root);
            }
            return P;
        }

        public static Polynomial QuadraticFactor(Tx root)
        {
            return new Polynomial(1, 2 * root.Real, root.Magnitude);
        }
```

```
        public static implicit operator DoubleDouble.Polynomial(DoubleComplex.Polynomial P)
        {
            return new DoubleDouble.Polynomial(P.ToArray());
        }
    }
}
```