```
namespace Simplex
{
    public class LPP
    {
        public ObjectiveFunction ObjFunc;
        public Constraint[] Constraints;
        public double[] Variables;

        public LPP(ObjectiveFunction objFunc, Constraint[] constraints)
        {
            this.ObjFunc = objFunc;
            this.Constraints = constraints;
            this.Variables = new double[ObjFunc.VariablesNumber];
        }

        public bool SolutionFound(Dictionary d)
        {
            return d.EntersBasis() == -1;
        }

        public void Solve()
        {
            Dictionary dict = new Dictionary(this);

            if (!dict.IsFeasible()) dict = this.initialize();

            Console.WriteLine("Finding solution...");
            Console.WriteLine("------------------------------");
            Console.WriteLine();

            while (!SolutionFound(dict))
            {
                dict.Print();
                dict.Improve();
            }
            dict.Print();

            for (int i = 0; i < dict.basic.Length; i++)
                if (dict.basic[i] < Variables.Length + 1)
                    Variables[dict.basic[i] - 1] = 0;
            for (int i = 0; i < dict.slack.Length; i++)
                if (dict.slack[i] < Variables.Length + 1)
                    Variables[dict.slack[i] - 1] = dict.c[i, 0];

        }

        private Dictionary initialize()
        {
            Console.WriteLine("Initialization phase...");
            Console.WriteLine("------------------------------");
            Console.WriteLine();

            double[] auxC = new double[ObjFunc.VariablesNumber + 1];
            auxC[0] = -1;
            for (int i = 0; i < auxC.Length - 1; i++) auxC[i + 1] = 0;
            ObjectiveFunction auxOF = new ObjectiveFunction(auxC);

            Constraint[] auxCS = new Constraint[this.Constraints.Length];
            int leavesBasis = 0;
            double minB = Constraints[0].Restriction;
            for (int i = 0; i < auxCS.Length; i++)
            {
                double[] auxCC = new double[ObjFunc.VariablesNumber + 1];
                auxCC[0] = -1;
                for (int j = 0; j < auxCC.Length - 1; j++)
                    auxCC[j + 1] = Constraints[i].Coefficients[j];
                auxCS[i] = new Constraint(auxCC, Constraints[i].Restriction);
                if (Constraints[i].Restriction < minB)
                    { minB = Constraints[i].Restriction; leavesBasis = i; }
```

```
        }

        LPP auxLPP = new LPP(auxOF, auxCS);
        Dictionary auxD = new Dictionary(auxLPP);
        auxD.Print(false);
        auxD.Recalculate(0, leavesBasis);
        while (!SolutionFound(auxD))
        {
            auxD.Print(preferToLeave: 1);
            auxD.Improve(preferToLeave: 1);
        }
        auxD.Print(preferToLeave: 1);

        int len = auxD.basic.Length;
        int[] bb = new int[len];
        double[,] cc = new double[auxLPP.Constraints.Length, len + 1];
        int[] ss = new int[auxLPP.Constraints.Length];
        for (int i = 0; i < len; i++)
        {
            bb[i] = auxD.basic[i];
            for (int j = 0; j < auxLPP.Constraints.Length; j++) cc[j, i + 1] = auxD.c[j, i + 1];
        }
        for (int j = 0; j < auxLPP.Constraints.Length; j++)
        { cc[j, 0] = auxD.c[j, 0]; ss[j] = auxD.slack[j]; }

        auxD.a = new double[len - 1];
        auxD.basic = new int[len - 1];
        auxD.slack = new int[auxLPP.Constraints.Length];
        auxD.c = new double[auxLPP.Constraints.Length, len];

        for (int i = 0; i < auxLPP.Constraints.Length; i++)
        {
            auxD.c[i, 0] = cc[i, 0];
            int j = 1;
            while (bb[j - 1] != 1) { auxD.c[i, j] = cc[i, j]; j++; }
            while (j < bb.Length) { j++; auxD.c[i, j - 1] = cc[i, j]; }
            auxD.slack[i] = ss[i] - 1;
        }
        int k = 0;
        while (bb[k] != 1) { auxD.basic[k] = bb[k] - 1; k++; }
        k++;
        while (k < bb.Length) { auxD.basic[k - 1] = bb[k] - 1; k++; }

        auxD.z0 = 0;
        for (int i = 0; i < this.ObjFunc.Coefficients.Length; i++)
            for (int j = 0; j < auxD.slack.Length; j++)
                if (auxD.slack[j] == i + 1)
                {
                    auxD.z0 += ObjFunc.Coefficients[i] * auxD.c[j, 0];
                    for (int m = 0; m < ObjFunc.Coefficients.Length; m++)
                        auxD.a[m] += ObjFunc.Coefficients[i] * auxD.c[j, m + 1];
                }
        auxD.Print(false);

        Console.WriteLine();

        return auxD;

    }
}
```

```
/// <summary>
/// Целевая функция
/// </summary>
public class ObjectiveFunction
{
    private double[] coefficients;

    public int VariablesNumber
    { get { return this.coefficients.Length; } }

    public double[] Coefficients
    { get { return this.coefficients; } }

    public ObjectiveFunction(double[] coefficients)
    {
        this.coefficients = coefficients;
    }

    public double Value(double[] variables)
    {
        double value = 0;

        if (VariablesNumber != variables.Length)
            throw new ArgumentException("The number of variables (" + variables.Length +
                                ") shoud be equal to the number of function coefficients (" +
                               this.VariablesNumber + ").");

        if (this.coefficients.Length > 0)
            for (int i = 0; i < VariablesNumber; i++)
                value += this.coefficients[i] * variables[i];

        return value;
    }
}


public class Constraint
{
    private double[] coefficients;
    private double restriction;
    private int coefficientsNumber;

    public double[] Coefficients
    { get { return this.coefficients; } }

    public double Restriction
    { get { return this.restriction; } }

    public Constraint(double[] coefficients, double restriction)
    {
        this.coefficientsNumber = coefficients.Length;
        this.coefficients = coefficients;
        this.restriction = restriction;
    }
}
```

```
public class Dictionary
{
    public double[,] c;
    public double[] a;
    public double z0;
    /// <summary>
    /// basic variables
    /// </summary>
    public int[] basic;
    /// <summary>
    /// slack variables
    /// </summary>
    public int[] slack;

    public Dictionary(LPP lpp)
    {
        z0 = 0;
        this.a = new double[lpp.ObjFunc.VariablesNumber];
        for (int i = 0; i < a.Length; i++)
            a[i] = lpp.ObjFunc.Coefficients[i];

        this.basic = new int[lpp.ObjFunc.VariablesNumber];
        for (int i = 0; i < lpp.ObjFunc.VariablesNumber; i++) this.basic[i] = i + 1;
        this.slack = new int[lpp.Constraints.Length];
        for (int i = 0; i < lpp.Constraints.Length; i++)
          this.slack[i] = lpp.ObjFunc.VariablesNumber + i + 1;

        this.c = new double[lpp.Constraints.Length, lpp.ObjFunc.VariablesNumber + 1];
        for (int i = 0; i < lpp.Constraints.Length; i++)
        {
            this.c[i, 0] = lpp.Constraints[i].Restriction;
            for (int j = 1; j < lpp.ObjFunc.VariablesNumber + 1; j++)
                this.c[i, j] = -lpp.Constraints[i].Coefficients[j - 1];
        }
    }

    public bool IsFeasible()
    {
        for (int i = 0; i < slack.Length; i++)
            if (c[i, 0] < 0) return false;
        return true;
    }

    /// <summary>
    /// Determines a variable to enter basis
    /// </summary>
    /// <returns>Index of a variable to enter basis. If there's no variable to enter, returns -1
        </returns>
    public int EntersBasis()
    {
        int n = -1;
        double maxA = this.a[0];
        if (maxA > 0) n = 0;

        for (int i = 0; i < a.Length; i++)
            if (a[i] > 0 && a[i] > maxA) { maxA = a[i]; n = i; }

        return n;
    }

    /// <summary>
    /// Determines a variable to leave basis
    /// </summary>
    /// <param name="enterIdx">Index of basic variable to enter basis</param>
    /// <returns>Index of a variable to leave basis. If there's no variable to leave, returns -1
        </returns>
    private int LeavesBasis(int enterIdx, int preferToLeave = -1)
    {
        if (enterIdx == -1) return -1;
```

```
    int idxPTL = -1;
    int n = -1;
    double[] dc = new double[slack.Length];

    for (int i = 0; i < dc.Length; i++)
    {
        if (preferToLeave != -1 && slack[i] == preferToLeave) idxPTL = i;
        if (c[i, 1 + enterIdx] < 0)
            dc[i] = c[i, 0] / c[i, 1 + enterIdx];
        else
            dc[i] = double.NegativeInfinity;
    }

    double maxDC = dc[0];
    if (maxDC > double.NegativeInfinity) n = 0;
    for (int i = 0; i < dc.Length; i++)
        if (dc[i] <= 0 && dc[i] >= maxDC)
        {
            maxDC = dc[i]; n = i;
            if (idxPTL != -1 && dc[idxPTL] == maxDC) n = idxPTL;
        }

    return n;
}

public void Recalculate(int enterIdx, int leaveIdx)
{
    // Recalculating coefficients for equation of entering variable
    c[leaveIdx, 0] = -c[leaveIdx, 0] / c[leaveIdx, enterIdx + 1];
    c[leaveIdx, enterIdx + 1] = 1 / c[leaveIdx, enterIdx + 1];
    for (int j = 0; j < basic.Length; j++)
        if (j != enterIdx)
            c[leaveIdx, j + 1] = -c[leaveIdx, j + 1] * c[leaveIdx, enterIdx + 1];

    // Recalculating coefficients for other equations
    for (int i = 0; i < slack.Length; i++)
        if (i != leaveIdx)
        {
            double oldC = c[i, enterIdx + 1];
            c[i, 0] = c[i, 0] + c[i, enterIdx + 1] * c[leaveIdx, 0];
            c[i, enterIdx + 1] = c[i, enterIdx + 1] * c[leaveIdx, enterIdx + 1];
            for (int j = 0; j < basic.Length; j++)
                if (j != enterIdx) c[i, j + 1] = c[i, j + 1] + oldC * c[leaveIdx, j + 1];

    // Recalculating coefficients for objective function
    z0 = z0 + a[enterIdx] * c[leaveIdx, 0];
    double oldA = a[enterIdx];
    a[enterIdx] = a[enterIdx] * c[leaveIdx, enterIdx + 1];
    for (int j = 0; j < basic.Length; j++)
        if (j != enterIdx) a[j] = a[j] + oldA * c[leaveIdx, j + 1];

    // Swaping names of basic and slack variables
    int valueToSwap = basic[enterIdx];
    basic[enterIdx] = slack[leaveIdx]; slack[leaveIdx] = valueToSwap;
}

public void Improve(int preferToLeave = -1)
{
    int eb = EntersBasis();
    if (eb != -1)
    {
        int lb = LeavesBasis(eb, preferToLeave);
        if (lb != -1)
            Recalculate(eb, lb);
    }
}
```

```
        public void Print(bool withAnalysis = true, int preferToLeave = -1)
        {
            Console.WriteLine();
            Console.WriteLine("Dictionary for LPP:");
            for (int i = 0; i < this.slack.Length; i++)
            {
                Console.Write("x{0} = {1} ", slack[i], c[i, 0]);
                for (int j = 0; j < this.a.Length; j++)
                    if (c[i, j + 1] < 0)
                        Console.Write("- {0}*x{1} ", -c[i, j + 1], basic[j]);
                    else
                        Console.Write("+ {0}*x{1} ", c[i, j + 1], basic[j]);
                Console.WriteLine();
            }
            Console.Write("z = {0} ", z0);
            for (int j = 0; j < this.a.Length; j++)
                if (a[j] < 0)
                    Console.Write("- {0}*x{1} ", -a[j], basic[j]);
                else
                    Console.Write("+ {0}*x{1} ", a[j], basic[j]);
            Console.WriteLine();

            if (withAnalysis)
            {
                int eb = EntersBasis();
                if (eb == -1)
                {
                    Console.WriteLine("No variables to enter basis - solution is found.");
                    Console.WriteLine("The optimal value of objective function is {0}.", z0);
                    Console.WriteLine("The optimal solution is:");
                    for (int i = 0; i < basic.Length; i++) Console.WriteLine("x{0} = 0", basic[i]);
                    for (int i = 0; i < slack.Length; i++)
                           Console.WriteLine("x{0} = {1}", slack[i], c[i, 0]);
                }
                else
                {
                    Console.WriteLine("Enters basis: x{0}", basic[eb]);
                    if (LeavesBasis(eb) == -1)
                        Console.WriteLine("No variables to leave basis.");
                    else
                        Console.WriteLine("Leaves basis: x{0}",
                                          slack[LeavesBasis(eb, preferToLeave)]);
                }
                Console.WriteLine();
            }
        }

    }

}
```