```csharp
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Binary mathematical operations.</para>
    /// <para>Бинарные математические операции.</para>
    /// </summary>
    public static class Arithmetic
    {

        /// <summary>
        /// <para>Adding the x and y arguments.</para>
        /// <para>Сложение аргументов x и y.</para>
        /// </summary>
        /// <param name = "x">
        /// <para>The first term.</para>
        /// <para>Первое слагаемое.</para>
        /// </param>
        /// <param name = "y">
        /// <para>The second term.</para>
        /// <para>Второе слагаемое.</para>
        /// </param>
        /// <returns>
        /// <para>Sum of x and y.</para>
        /// <para>Сумма x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);

        /// <summary>
        /// <para>Subtracting x from y.</para>
        /// <para>Вычитание x из y.</para>
        /// </summary>
        /// <param name = "x">
        /// <para>Minuend.</para>
        /// <para>Уменьшаемое.</para>
        /// </param>
        /// <param name = "y">
        /// <para>Subtrahend.</para>
        /// <para>Вычитаемое.</para>
        /// </param>
        /// <returns>
        /// <para>Difference of x and y.</para>
        /// <para>Разность x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);

        /// <summary>
        /// <para>Multiplication the x and y.</para>
        /// <para>Умножение x и y.</para>
        /// </summary>
        /// <param name = "x">
        /// <para>First multiplier.</para>
        /// <para>Первый множитель.</para>
        /// </param>
        /// <param name = "y">
        /// <para>Second multiplier.</para>
        /// <para>Второй множитель.</para>
        /// </param>
        /// <returns>
        /// <para>Product of x and y.</para>
        /// <para>Произведение x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);

        /// <summary>
        /// <para>Dividing x by y.</para>
        /// <para>Деление x на y.</para>
        /// </summary>
        /// <param name = "x">
        /// <para>Dividend.</para>
```

```csharp
 77            /// <para>Делимое.</para>
 78            /// </param>
 79            /// <param name = "y">
 80            /// <para>Divider.</para>
 81            /// <para>Делитель.</para>
 82            /// </param>
 83            /// <returns>
 84            /// <para>Quoitent of x and y.</para>
 85            /// <para>Частное x и y.</para>
 86            /// </returns>
 87            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 88            public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);
 89
 90            /// <summary>
 91            /// <para>Increasing the parameter x by one.</para>
 92            /// <para>Увеличение параметра x на единицу.</para>
 93            /// </summary>
 94            /// <param name = "x">
 95            /// <para>Increase the number required.</para>
 96            /// <para>Число необходимое увеличить.</para>
 97            /// </param>
 98            /// <returns>
 99            /// <para>Increase by one x.</para>
100            /// <para>Увеличенное на единицу x.</para>
101            /// </returns>
102            [MethodImpl(MethodImplOptions.AggressiveInlining)]
103            public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
104
105            /// <summary>
106            /// <para>Increase the parameter x passed by reference by one.</para>
107            /// <para>Увеличение переданного по ссылке параметра x на единицу.</para>
108            /// </summary>
109            /// <param name = "x">
110            /// <para>Increase the argument required.</para>
111            /// <para>Аргумент требуемый увеличить.</para>
112            /// </param>
113            /// <returns>
114            /// <para>Increase by one x with returning the value to the original variable.</para>
115            /// <para>Увеличенное на единицу x с возвратом значения в исходную переменную.</para>
116            /// </returns>
117            [MethodImpl(MethodImplOptions.AggressiveInlining)]
118            public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);
119
120            /// <summary>
121            /// <para>Decrease parameter x by one.</para>
122            /// <para>Уменьшение параметра x на единицу.</para>
123            /// </summary>
124            /// <param name = "x">
125            /// <para>Increase the number reduce.</para>
126            /// <para>Число необходимое уменьшить.</para>
127            /// </param>
128            /// <returns>
129            /// <para>Reduced by one x.</para>
130            /// <para>Уменьшенное на единицу x.</para>
131            /// </returns>
132            [MethodImpl(MethodImplOptions.AggressiveInlining)]
133            public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
134
135            /// <summary>
136            /// <para>Decreasing the parameter x passed by reference by one.</para>
137            /// <para>Уменьшение переданного по ссылке параметра x на единицу.</para>
138            /// </summary>
139            /// <param name = "x">
140            /// <para>Increase the argument reduce.</para>
141            /// <para>Аргумент требуемый уменьшить.</para>
142            /// </param>
143            /// <returns>
144            /// <para>Reduced by one x with returning the value to the original variable.</para>
145            /// <para>Уменьшенное на единицу x с воз вратом значения в исходную переменную.</para>
146            /// </returns>
147            [MethodImpl(MethodImplOptions.AggressiveInlining)]
148            public static T Decrement<T>(ref T x) => x = Arithmetic<T>.Decrement(x);
149        }
150    }
```

1.2   ./csharp/Platform.Numbers/ArithmeticExtensions.cs

```csharp
 1  using System.Runtime.CompilerServices;
 2
```

```csharp
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    public static class ArithmeticExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Decrement<T>(this ref T x) where T : struct => x =
        ↪  Arithmetic<T>.Decrement(x);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(this ref T x) where T : struct => x =
        ↪  Arithmetic<T>.Increment(x);
    }
}
```

1.3   ./csharp/Platform.Numbers/Arithmetic[T].cs

```csharp
using System;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;

// ReSharper disable StaticFieldInGenericType
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    public static class Arithmetic<T>
    {
        public static readonly Func<T, T, T> Add = CompileAddDelegate();
        public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();
        public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();
        public static readonly Func<T, T, T> Divide = CompileDivideDelegate();
        public static readonly Func<T, T> Increment = CompileIncrementDelegate();
        public static readonly Func<T, T> Decrement = CompileDecrementDelegate();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileAddDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Add();
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileSubtractDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Subtract();
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileMultiplyDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Emit(OpCodes.Mul);
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileDivideDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
```

```
63              emiter.LoadArguments(0, 1);
64              if(NumericType<T>.IsSigned)
65              {
66                  emiter.Emit(OpCodes.Div);
67              }
68              else
69              {
70                  emiter.Emit(OpCodes.Div_Un);
71              }
72              emiter.Return();
73          });
74      }
75
76      [MethodImpl(MethodImplOptions.AggressiveInlining)]
77      private static Func<T, T> CompileIncrementDelegate()
78      {
79          return DelegateHelpers.Compile<Func<T, T>>(emiter =>
80          {
81              Ensure.Always.IsNumeric<T>();
82              emiter.LoadArgument(0);
83              emiter.Increment<T>();
84              emiter.Return();
85          });
86      }
87
88      [MethodImpl(MethodImplOptions.AggressiveInlining)]
89      private static Func<T, T> CompileDecrementDelegate()
90      {
91          return DelegateHelpers.Compile<Func<T, T>>(emiter =>
92          {
93              Ensure.Always.IsNumeric<T>();
94              emiter.LoadArgument(0);
95              emiter.Decrement<T>();
96              emiter.Return();
97          });
98      }
99  }
100 }
```

1.4 ./csharp/Platform.Numbers/Bit.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5
6   namespace Platform.Numbers
7   {
8       /// <summary>
9       /// <para>A set of operations on the set bits of a number.</para>
10      /// <para>Набор операций над установленными битами числа.</para>
11      /// </summary>
12      public static class Bit
13      {
14          /// <summary>
15          /// <para>Counts the number of bits set in a number.</para>
16          /// <para>Подсчитывает количество установленных бит в числе.</para>
17          /// </summary>
18          /// <param>
19          /// <para>Bitwise number.</para>
20          /// <para>Число в битовом представлении.</para>
21          /// </param>
22          /// <returns>
23          /// <para>Number of bits set in a number.</para>
24          /// <para>Количество установленных бит в числе.</para>
25          /// </returns>
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public static long Count(long x)
28          {
29              long n = 0;
30              while (x != 0)
31              {
32                  n++;
33                  x &= x - 1;
34              }
35              return n;
36          }
37          /// <summary>
```

```csharp
        /// <para>Searches for the first bit set in a number.</para>
        /// <para>Ищет первый установленный бит в числе.</para>
        /// </summary>
        /// <param>
        /// <para>Bitwise number.</para>
        /// <para>Число в битовом представлении.</para>
        /// </param>
        /// <returns>
        /// <para>First bit set.</para>
        /// <para>Первый установленный бит.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GetLowestPosition(ulong value)
        {
            if (value == 0)
            {
                return -1;
            }
            var position = 0;
            while ((value & 1UL) == 0)
            {
                value >>= 1;
                ++position;
            }
            return position;
        }

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Not<T>(T x) => Bit<T>.Not(x);

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T And<T>(T x, T y) => Bit<T>.And(x, y);

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
```

```
118        /// </returns>
119        [MethodImpl(MethodImplOptions.AggressiveInlining)]
120        public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
121
122        /// <summary>
123        /// <para>.</para>
124        /// <para>.</para>
125        /// </summary>
126        /// <returns>
127        /// <para>.</para>
128        /// <para>.</para>
129        /// </returns>
130        [MethodImpl(MethodImplOptions.AggressiveInlining)]
131        public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
           ↪  Bit<T>.PartialWrite(target, source, shift, limit);
132
133        /// <summary>
134        /// <para>.</para>
135        /// <para>.</para>
136        /// </summary>
137        /// <returns>
138        /// <para>.</para>
139        /// <para>.</para>
140        /// </returns>
141        [MethodImpl(MethodImplOptions.AggressiveInlining)]
142        public static T PartialRead<T>(T target, int shift, int limit) =>
           ↪  Bit<T>.PartialRead(target, shift, limit);
143    }
144 }
```

1.5 ./csharp/Platform.Numbers/BitExtensions.cs

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Numbers
6  {
7      /// <summary>
8      /// <returns> bit operations </returns>
9      /// </summary>
10     /* bit operations */
11     public static class BitwiseExtensions
12
13     {
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
           ↪  T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static T PartialRead<T>(this T target, int shift, int limit) =>
           ↪  Bit<T>.PartialRead(target, shift, limit);
23     }
24 }
```

1.6 ./csharp/Platform.Numbers/Bit[T].cs

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Exceptions;
4  using Platform.Reflection;
5
6  // ReSharper disable StaticFieldInGenericType
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Numbers
10 {
11     public static class Bit<T>
12     {
13         public static readonly Func<T, T> Not = CompileNotDelegate();
14         public static readonly Func<T, T, T> Or = CompileOrDelegate();
15         public static readonly Func<T, T, T> And = CompileAndDelegate();
16         public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();
17         public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();
18         public static readonly Func<T, T, int, int, T> PartialWrite =
           ↪  CompilePartialWriteDelegate();
19         public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T> CompileNotDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0);
                emiter.Not();
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileOrDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Or();
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileAndDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.And();
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, int, T> CompileShiftLeftDelegate()
        {
            return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.ShiftLeft();
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, int, T> CompileShiftRightDelegate()
        {
            return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.ShiftRight();
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                var constants = GetConstants();
                var bitsNumber = constants.Item1;
                var numberFilledWithOnes = constants.Item2;
                ushort shiftArgument = 2;
                ushort limitArgument = 3;
                var checkLimit = emiter.DefineLabel();
                var calculateSourceMask = emiter.DefineLabel();
                // Check shift
                emiter.LoadArgument(shiftArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
```

```csharp
                    // Fix shift
                    emiter.LoadConstant(bitsNumber);
                    emiter.LoadArgument(shiftArgument);
                    emiter.Add();
                    emiter.StoreArgument(shiftArgument);
                    emiter.MarkLabel(checkLimit);
                    // Check limit
                    emiter.LoadArgument(limitArgument);
                    emiter.LoadConstant(0);
                    emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
                    // Fix limit
                    emiter.LoadConstant(bitsNumber);
                    emiter.LoadArgument(limitArgument);
                    emiter.Add();
                    emiter.StoreArgument(limitArgument);
                    emiter.MarkLabel(calculateSourceMask);
                    var sourceMask = emiter.DeclareLocal<T>();
                    var targetMask = emiter.DeclareLocal<T>();
                    emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                    emiter.LoadArgument(limitArgument);
                    emiter.ShiftLeft();
                    emiter.Not();
                    emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                    emiter.And();
                    emiter.StoreLocal(sourceMask);
                    emiter.LoadLocal(sourceMask);
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftLeft();
                    emiter.Not();
                    emiter.StoreLocal(targetMask);
                    emiter.LoadArgument(0); // target
                    emiter.LoadLocal(targetMask);
                    emiter.And();
                    emiter.LoadArgument(1); // source
                    emiter.LoadLocal(sourceMask);
                    emiter.And();
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftLeft();
                    emiter.Or();
                    emiter.Return();
                });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, int, int, T> CompilePartialReadDelegate()
        {
            return DelegateHelpers.Compile<Func<T, int, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                var constants = GetConstants();
                var bitsNumber = constants.Item1;
                var numberFilledWithOnes = constants.Item2;
                ushort shiftArgument = 1;
                ushort limitArgument = 2;
                var checkLimit = emiter.DefineLabel();
                var calculateSourceMask = emiter.DefineLabel();
                // Check shift
                emiter.LoadArgument(shiftArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
                // Fix shift
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(shiftArgument);
                emiter.Add();
                emiter.StoreArgument(shiftArgument);
                emiter.MarkLabel(checkLimit);
                // Check limit
                emiter.LoadArgument(limitArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
                // Fix limit
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(limitArgument);
                emiter.Add();
                emiter.StoreArgument(limitArgument);
                emiter.MarkLabel(calculateSourceMask);
                var sourceMask = emiter.DeclareLocal<T>();
                var targetMask = emiter.DeclareLocal<T>();
```

```csharp
                    emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                    emiter.LoadArgument(limitArgument); // limit
                    emiter.ShiftLeft();
                    emiter.Not();
                    emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                    emiter.And();
                    emiter.StoreLocal(sourceMask);
                    emiter.LoadLocal(sourceMask);
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftLeft();
                    emiter.StoreLocal(targetMask);
                    emiter.LoadArgument(0); // target
                    emiter.LoadLocal(targetMask);
                    emiter.And();
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftRight();
                    emiter.Return();
                });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Tuple<int, T> GetConstants()
        {
            var type = typeof(T);
            if (type == typeof(ulong))
            {
                return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
            }
            if (type == typeof(uint))
            {
                return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
            }
            if (type == typeof(ushort))
            {
                return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
            }
            if (type == typeof(byte))
            {
                return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
            }
            throw new NotSupportedException();
        }
    }
}
```

1.7 ./csharp/Platform.Numbers/Math.cs

```csharp
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <remarks>
    /// Resizable array (FileMappedMemory) for values cache may be used. or cached oeis.org
    /// </remarks>
    public static class Math
    {
        /// <remarks>
        /// <para>Source: https://oeis.org/A000142/list </para>
        /// <para>Источник: https://oeis.org/A000142/list </para>
        /// </remarks>

        private static readonly ulong[] _factorials =
        {
            1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
            479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
            355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
        };

        /// <remarks>
        /// <para>Source: https://oeis.org/A000108/list </para>
        /// <para>Источник: https://oeis.org/A000108/list </para>
        /// </remarks>
        private static readonly ulong[] _catalans =
        {
            1,    1,    2,    5,    14,    42,    132,    429,    1430,    4862,    16796,    58786,    208012,
            742900,    2674440,    9694845,    35357670,    129644790,    477638700,    1767263190,
```

```
33          6564120420,   24466267020,   91482563640,   343059613650,   1289904147324,
       ↪   4861946401452,
34          18367353072152,  69533550916004,  26374951750360,  1002242216651368,
       ↪   3814986502092304,
35          14544636039226909, 5553406487704898, 212336130412243110, 812944042149730764,
       ↪   3116285494907301262, 11959798385860453492
36      };
37
38      public static readonly ulong MaximumFactorialNumber = 20;
39
40      public static readonly ulong MaximumCatalanIndex = 36;
41
42      /// <summary>
43      /// <para>Returns the product of all positive integers less than or equal to the number
       ↪   specified as an argument.</para>
44      /// <para>Возвращает произведение всех положительных чисел меньше или равных указанному
       ↪   в качестве аргумента числу.</para>
45      /// </summary>
46      /// <param name="n"><para>The maximum positive number that will participate in
       ↪   factorial's product.</para><para>Максимальное положительное число, которое будет
       ↪   участвовать в произведение факториала.</para></param>
47      /// <returns><para>The product of all positive integers less than or equal to the number
       ↪   specified as an argument.</para><para>Произведение всех положительных чисел меньше
       ↪   или равных указанному в качестве аргумента числу.</para></returns>
48      public static ulong Factorial(ulong n)
49      {
50          if (n >= 0 && n <= MaximumFactorialNumber)
51          {
52              return _factorials[n];
53          }
54          else
55          {
56              throw new ArgumentOutOfRangeException($"Only numbers from 0 to
       ↪   {MaximumFactorialNumber} are supported by unsigned integer with 64 bits
       ↪   length.");
57          }
58      }
59
60      /// <summary>
61      /// <para>Returns the Catalan Number with the number specified as an argument.</para>
62      /// <para>Возвращает Каталановое число с номером указанным в качестве аргумента.</para>
63      /// </summary>
64      /// <param name="n"><para>The number of Catalan number.</para><para>Номер Каталанового
       ↪   числа.</para></param>
65      /// <returns><para>The Catalan Number with the number specified as an
       ↪   argument.</para><para>Каталановое число с номером указанным в качестве
       ↪   аргумента.</para></returns>
66      public static ulong Catalan(ulong n)
67      {
68          if (n >= 0 && n <= MaximumCatalanIndex)
69          {
70              return _catalans[n];
71          }
72          else
73          {
74              throw new ArgumentOutOfRangeException($"Only numbers from 0 to
       ↪   {MaximumCatalanIndex} are supported by unsigned integer with 64 bits
       ↪   length.");
75          }
76      }
77
78      [MethodImpl(MethodImplOptions.AggressiveInlining)]
79      public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
80
81      [MethodImpl(MethodImplOptions.AggressiveInlining)]
82      public static T Abs<T>(T x) => Math<T>.Abs(x);
83
84      [MethodImpl(MethodImplOptions.AggressiveInlining)]
85      public static T Negate<T>(T x) => Math<T>.Negate(x);
86    }
87  }
```

1.8 ./csharp/Platform.Numbers/MathExtensions.cs

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Numbers
```

```
 6  {
 7      /// <summary>
 8      /// <para>Provides a set of extension methods that perform mathematical operations on
        ↪  arbitrary object types.</para>
 9      /// <para>Предоставляет набор методов расширения выполняющих математические операции для
        ↪  объектов произвольного типа.</para>
10      /// </summary>
11      public static class MathExtensions
12      {
13          /// <summary>
14          /// <para>Takes a module from a number.</para>
15          /// <para>Берёт модуль от числа.</para>
16          /// </summary>
17          /// <param name = "x">
18          /// <para>The number from which to take the absolute value.</para>
19          /// <para>Число от которого необходимо взять абсолютное значение.</para>
20          /// </param>
21          /// <returns>
22          /// <para>The absolute value of a number.</para>
23          /// <para>Абсолютное значение числа.</para>
24          /// </returns>
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);
27
28          /// <summary>
29          /// <para>Makes a number negative.</para>
30          /// <para>Делает число отрицательным.</para>
31          /// </summary>
32          /// <param name = "x">
33          /// <para>The number to be made negative.</para>
34          /// <para>Число которое нужно сделать отрицательным.</para>
35          /// </param>
36          /// <returns>
37          /// <para>Negative number.</para>
38          /// <para>Отрицательное число.</para>
39          /// </returns>
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
42      }
43  }
```

1.9   ./csharp/Platform.Numbers/Math[T].cs

```
 1  using System;
 2  using System.Runtime.CompilerServices;
 3  using Platform.Exceptions;
 4  using Platform.Reflection;
 5
 6  // ReSharper disable StaticFieldInGenericType
 7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 8
 9  namespace Platform.Numbers
10  {
11      public static class Math<T>
12      {
13          public static readonly Func<T, T> Abs = CompileAbsDelegate();
14          public static readonly Func<T, T> Negate = CompileNegateDelegate();
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          private static Func<T, T> CompileAbsDelegate()
18          {
19              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
20              {
21                  Ensure.Always.IsNumeric<T>();
22                  emiter.LoadArgument(0);
23                  if (NumericType<T>.IsSigned)
24                  {
25                      emiter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
26                  }
27                  emiter.Return();
28              });
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          private static Func<T, T> CompileNegateDelegate()
33          {
34              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
35              {
36                  emiter.LoadArgument(0);
37                  emiter.Negate();
```

```
38            emiter.Return();
39          });
40      }
41   }
42 }
```

1.10   ./csharp/Platform.Numbers.Tests/ArithmeticExtensionsTests.cs

```
1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      public static class ArithmeticExtensionsTests
6      {
7          [Fact]
8          public static void IncrementTest()
9          {
10             var number = 0UL;
11             var returnValue = number.Increment();
12             Assert.Equal(1UL, returnValue);
13             Assert.Equal(1UL, number);
14         }
15
16         [Fact]
17         public static void DecrementTest()
18         {
19             var number = 1UL;
20             var returnValue = number.Decrement();
21             Assert.Equal(0UL, returnValue);
22             Assert.Equal(0UL, number);
23         }
24     }
25 }
```

1.11   ./csharp/Platform.Numbers.Tests/ArithmeticTests.cs

```
1  using System;
2  using Xunit;
3
4  namespace Platform.Numbers.Tests
5  {
6      public static class ArithmeticTests
7      {
8          [Fact]
9          public static void CompiledOperationsTest()
10         {
11             Assert.Equal(3, Arithmetic.Add(1, 2));
12             Assert.Equal(1, Arithmetic.Subtract(2, 1));
13             Assert.Equal(8, Arithmetic.Multiply(2, 4));
14             Assert.Equal(4, Arithmetic.Divide(8, 2));
15             Assert.Equal(2, Arithmetic.Increment(1));
16             Assert.Equal(1UL, Arithmetic.Decrement(2UL));
17             Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
18         }
19     }
20 }
```

1.12   ./csharp/Platform.Numbers.Tests/BitTests.cs

```
1  using System;
2  using Xunit;
3
4  namespace Platform.Numbers.Tests
5  {
6      public static class BitTests
7      {
8          [Theory]
9          [InlineData(00, -1)] // 0000 0000 (none, -1)
10         [InlineData(01, 00)] // 0000 0001 (first, 0)
11         [InlineData(08, 03)] // 0000 1000 (forth, 3)
12         [InlineData(88, 03)] // 0101 1000 (forth, 3)
13         public static void GetLowestBitPositionTest(ulong value, int expectedPosition)
14         {
15             Assert.True(Bit.GetLowestPosition(value) == expectedPosition);
16         }
17
18         [Fact]
19         public static void ByteBitwiseOperationsTest()
20         {
21             Assert.True(Bit<byte>.Not(2) == unchecked((byte)~2));
22             Assert.True(Bit<byte>.Or(1, 2) == (1 | 2));
```

```csharp
                    Assert.True(Bit<byte>.And(1, 2) == (1 & 2));
                    Assert.True(Bit<byte>.ShiftLeft(1, 2) == (1 << 2));
                    Assert.True(Bit<byte>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void UInt16BitwiseOperationsTest()
        {
                    Assert.True(Bit<ushort>.Not(2) == unchecked((ushort)~2));
                    Assert.True(Bit<ushort>.Or(1, 2) == (1 | 2));
                    Assert.True(Bit<ushort>.And(1, 2) == (1 & 2));
                    Assert.True(Bit<ushort>.ShiftLeft(1, 2) == (1 << 2));
                    Assert.True(Bit<ushort>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void UInt32BitwiseOperationsTest()
        {
                    Assert.True(Bit<uint>.Not(2) == unchecked((uint)~2));
                    Assert.True(Bit<uint>.Or(1, 2) == (1 | 2));
                    Assert.True(Bit<uint>.And(1, 2) == (1 & 2));
                    Assert.True(Bit<uint>.ShiftLeft(1, 2) == (1 << 2));
                    Assert.True(Bit<uint>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void UInt64BitwiseOperationsTest()
        {
                    Assert.True(Bit<ulong>.Not(2) == unchecked((ulong)~2));
                    Assert.True(Bit<ulong>.Or(1, 2) == (1 | 2));
                    Assert.True(Bit<ulong>.And(1, 2) == (1 & 2));
                    Assert.True(Bit<ulong>.ShiftLeft(1, 2) == (1 << 2));
                    Assert.True(Bit<ulong>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void PartialReadWriteTest()
        {
            {
                    uint firstValue = 1;
                    uint secondValue = 1543;

                    // Pack (join) two values at the same time
                    uint value = secondValue << 1 | firstValue;

                    uint unpackagedFirstValue = value & 1;
                    uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                    Assert.True(firstValue == unpackagedFirstValue);
                    Assert.True(secondValue == unpackagedSecondValue);

                    // Using universal functions:
                    Assert.True(PartialRead(value, 0, 1) == firstValue);
                    Assert.True(PartialRead(value, 1, -1) == secondValue);

                    firstValue = 0;
                    secondValue = 6892;

                    value = PartialWrite(value, firstValue, 0, 1);
                    value = PartialWrite(value, secondValue, 1, -1);

                    Assert.True(PartialRead(value, 0, 1) == firstValue);
                    Assert.True(PartialRead(value, 1, -1) == secondValue);
            }

            {
                    uint firstValue = 1;
                    uint secondValue = 1543;

                    // Pack (join) two values at the same time
                    uint value = secondValue << 1 | firstValue;

                    uint unpackagedFirstValue = value & 1;
                    uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                    Assert.True(firstValue == unpackagedFirstValue);
                    Assert.True(secondValue == unpackagedSecondValue);

                    // Using universal functions:
                    Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
```

```csharp
                Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);

                firstValue = 0;
                secondValue = 6892;

                value = Bit.PartialWrite(value, firstValue, 0, 1);
                value = Bit.PartialWrite(value, secondValue, 1, -1);

                Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
                Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
            }

            {
                uint firstValue = 1;
                uint secondValue = 1543;

                // Pack (join) two values at the same time
                uint value = secondValue << 1 | firstValue;

                uint unpackagedFirstValue = value & 1;
                uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                Assert.True(firstValue == unpackagedFirstValue);
                Assert.True(secondValue == unpackagedSecondValue);

                // Using universal functions:
                var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
                var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
                var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
                var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);

                Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);

                firstValue = 0;
                secondValue = 6892;

                value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
                value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);

                Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
            }
        }

        // TODO: Can be optimized using precalculation of TargetMask and SourceMask
        private static uint PartialWrite(uint target, uint source, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return target & targetMask | (source & sourceMask) << shift;
        }

        private static uint PartialRead(uint target, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return (target & targetMask) >> shift;
        }

        private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
        {
            if (shift < 0)
            {
```

```
183            shift = 32 + shift;
184        }
185        if (limit < 0)
186        {
187            limit = 32 + limit;
188        }
189        var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
190        var targetMask = ~(sourceMask << shift);
191        return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
192    }
193
194    private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
195    {
196        if (shift < 0)
197        {
198            shift = 32 + shift;
199        }
200        if (limit < 0)
201        {
202            limit = 32 + limit;
203        }
204        var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
205        var targetMask = sourceMask << shift;
206        return new Tuple<uint, int>(targetMask, shift);
207    }
208
209    private static uint PartialWrite(uint target, uint targetMask, uint source, uint
    ↪ sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
210
211    private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
    ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
    ↪ masksAndShift.Item2, masksAndShift.Item3);
212
213    private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
    ↪ targetMask) >> shift;
214
215    private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
    ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
216
217    [Fact]
218    public static void BugWithLoadingConstantOf8Test()
219    {
220        Bit<byte>.PartialWrite(0, 1, 5, -5);
221    }
222  }
223 }
```

1.13 ./csharp/Platform.Numbers.Tests/MathExtensionsTests.cs

```
1 using Xunit;
2
3 namespace Platform.Numbers.Tests
4 {
5     public static class MathExtensionsTests
6     {
7         [Fact]
8         public static void AbsTest()
9         {
10            var number = -1L;
11            var returnValue = number.Abs();
12            Assert.Equal(1L, returnValue);
13            Assert.Equal(1L, number);
14        }
15
16        [Fact]
17        public static void NegateTest()
18        {
19            var number = 2L;
20            var returnValue = number.Negate();
21            Assert.Equal(-2L, returnValue);
22            Assert.Equal(-2L, number);
23        }
24
25        [Fact]
26        public static void UnsignedNegateTest()
27        {
28            var number = 2UL;
29            var returnValue = number.Negate();
30            Assert.Equal(18446744073709551614, returnValue);
31            Assert.Equal(18446744073709551614, number);
```

```
32              }
33          }
34      }
```

1.14 ./csharp/Platform.Numbers.Tests/MathTests.cs

```
1   using Xunit;
2
3   namespace Platform.Numbers.Tests
4   {
5       public static class MathTests
6       {
7           [Fact]
8           public static void CompiledOperationsTest()
9           {
10              Assert.True(Math.Abs(Arithmetic<double>.Subtract(3D, 2D) - 1D) < 0.01);
11          }
12      }
13  }
```

1.15 ./csharp/Platform.Numbers.Tests/SystemTests.cs

```
1   using Xunit;
2
3   namespace Platform.Numbers.Tests
4   {
5       public static class SystemTests
6       {
7           [Fact]
8           public static void PossiblePackTwoValuesIntoOneTest()
9           {
10              uint value = 0;
11
12              // Set one to first bit
13              value |= 1;
14
15              Assert.True(value == 1);
16
17              // Set zero to first bit
18              value &= 0xFFFFFFFE;
19
20              // Get first bit
21              uint read = value & 1;
22
23              Assert.True(read == 0);
24
25              uint firstValue = 1;
26              uint secondValue = 1543;
27
28              // Pack (join) two values at the same time
29              value = (secondValue << 1) | firstValue;
30
31              uint unpackagedFirstValue = value & 1;
32              uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;
33
34              Assert.True(firstValue == unpackagedFirstValue);
35              Assert.True(secondValue == unpackagedSecondValue);
36
37              // Using universal functions:
38
39              Assert.True(PartialRead(value, 0, 1) == firstValue);
40              Assert.True(PartialRead(value, 1, -1) == secondValue);
41
42              firstValue = 0;
43              secondValue = 6892;
44
45              value = PartialWrite(value, firstValue, 0, 1);
46              value = PartialWrite(value, secondValue, 1, -1);
47
48              Assert.True(PartialRead(value, 0, 1) == firstValue);
49              Assert.True(PartialRead(value, 1, -1) == secondValue);
50          }
51
52          private static uint PartialWrite(uint target, uint source, int shift, int limit)
53          {
54              if (shift < 0)
55              {
56                  shift = 32 + shift;
57              }
58              if (limit < 0)
59              {
60                  limit = 32 + limit;
```

```csharp
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return (target & targetMask) | ((source & sourceMask) << shift);
        }

        private static uint PartialRead(uint target, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return (target & targetMask) >> shift;
        }
    }
}
```

Index