

algorithm.cm

```
/*  
  
    Copyright (c) 2012–2015 Seppo Laakko  
    http://sourceforge.net/projects/cmajor/  
  
    Distributed under the GNU General Public License, version 3 (GPLv3).  
    (See accompanying LICENSE.txt or http://www.gnu.org/licenses/gpl.html  
    )  
  
*/  
  
// Copyright (c) 1994  
// Hewlett–Packard Company  
// Copyright (c) 1996  
// Silicon Graphics Computer Systems, Inc.  
// Copyright (c) 2009 Alexander Stepanov and Paul McJones  
  
using System.Collections;  
using System.Concepts;  
  
namespace System  
{  
    public nothrow inline const T& Min<T>(const T& left, const T& right)  
        where T is LessThanComparable  
    {  
        if (left <= right) return left;  
        return right;  
    }  
  
    public nothrow inline const T& Max<T>(const T& left, const T& right)  
        where T is LessThanComparable  
    {  
        if (right >= left) return right;  
        return left;  
    }  
  
    public nothrow inline void Swap<T>(T& left, T& right) where T is  
        MoveConstructible and T is MoveAssignable and T is Destructible  
    {  
        T temp(Rvalue(left));  
        left = Rvalue(right);  
        right = Rvalue(temp);  
    }  
  
    public nothrow void Reverse<I>(I begin, I end) where I is  
        RandomAccessIterator  
    {  
        while (begin < end)  
        {
```

```

        --end;
        Swap(*begin, *end);
        ++begin;
    }
}

public nothrow void Reverse<I>(I begin, I end) where I is
    BidirectionalIterator
{
    while (true)
    {
        if (begin == end)
        {
            return;
        }
        --end;
        if (begin == end)
        {
            return;
        }
        Swap(*begin, *end);
        ++begin;
    }
}

public Pair<I, I> ReverseUntil<I>(I first, I middle, I last) where I
    is BidirectionalIterator
{
    while (first != middle && middle != last)
    {
        --last;
        Swap(*first, *last);
        ++first;
    }
    return MakePair(first, last);
}

public I Rotate<I>(I first, I middle, I last) where I is
    BidirectionalIterator
{
    Reverse(first, middle);
    Reverse(middle, last);
    Pair<I, I> p = ReverseUntil(first, middle, last);
    Reverse(p.first, p.second);
    if (middle == p.first) return p.second;
    return p.first;
}

public O Copy<I, O>(I begin, I end, O to) where I is InputIterator
    and O is OutputIterator and CopyAssignable<O.ValueType, I.
    ValueType>
{
    while (begin != end)

```

```

    {
        *to = *begin;
        ++begin;
        ++to;
    }
    return to;
}

public O CopyBackward<I, O>(I begin, I end, O to) where I is
    BidirectionalIterator and O is BidirectionalIterator and
    CopyAssignable<O.ValueType, I.ValueType>
{
    while (begin != end)
    {
        --to;
        --end;
        *to = *end;
    }
    return to;
}

public O Move<I, O>(I begin, I end, O to) where I is InputIterator
    and O is OutputIterator and O.ValueType is I.ValueType and I.
    ValueType is MoveAssignable
{
    while (begin != end)
    {
        *to = Rvalue(*begin);
        ++begin;
        ++to;
    }
    return to;
}

public O MoveBackward<I, O>(I begin, I end, O to) where I is
    BidirectionalIterator and O is BidirectionalIterator and O.
    ValueType is I.ValueType and I.ValueType is MoveAssignable
{
    while (begin != end)
    {
        --to;
        --end;
        *to = Rvalue(*end);
    }
    return to;
}

public nothrow int Distance<I>(I first, I last) where I is
    ForwardIterator
{
    int distance = 0;
    while (first != last)
    {

```

```

        ++first;
        ++distance;
    }
    return distance;
}

public nothrow inline int Distance<I>(I first , I last) where I is
    RandomAccessIterator
{
    return last - first;
}

public nothrow I Next<I>(I i, int n) where I is ForwardIterator
{
    #assert(n >= 0);
    while (n > 0)
    {
        ++i;
        --n;
    }
    return i;
}

public nothrow inline I Next<I>(I i, int n) where I is
    RandomAccessIterator
{
    return i + n;
}

public nothrow I LowerBound<I, T>(I first , I last , const T& value)
    where I is ForwardIterator and TotallyOrdered<T, I.ValueType>
{
    int len = Distance(first , last);
    while (len > 0)
    {
        int half = len >> 1;
        I middle = Next(first , half);
        if (value > *middle)
        {
            first = middle;
            ++first;
            len = len - half - 1;
        }
        else // value <= *middle
        {
            len = half;
        }
    }
    return first;
}

public nothrow I LowerBound<I, T, R>(I first , I last , const T& value ,
    R r) where I is ForwardIterator and T is I.ValueType and R is

```

```

    Relation and R.Domain is I.ValueType
{
    int len = Distance(first , last);
    while (len > 0)
    {
        int half = len >> 1;
        I middle = Next(first , half);
        if (r(*middle, value)) // value > *middle
        {
            first = middle;
            ++first;
            len = len - half - 1;
        }
        else // value <= *middle
        {
            len = half;
        }
    }
    return first;
}

public nothrow I UpperBound<I, T>(I first , I last , const T& value)
    where I is ForwardIterator and TotallyOrdered<T, I.ValueType>
{
    int len = Distance(first , last);
    while (len > 0)
    {
        int half = len >> 1;
        I middle = Next(first , half);
        if (value < *middle)
        {
            len = half;
        }
        else // value >= *middle
        {
            first = middle;
            ++first;
            len = len - half - 1;
        }
    }
    return first;
}

public nothrow I UpperBound<I, T, R>(I first , I last , const T& value ,
    R r) where I is ForwardIterator and T is I.ValueType and R is
    Relation and R.Domain is I.ValueType
{
    int len = Distance(first , last);
    while (len > 0)
    {
        int half = len >> 1;
        I middle = Next(first , half);
        if (r(value , *middle)) // value < *middle

```

```

        {
            len = half;
        }
        else // value >= *middle
        {
            first = middle;
            ++first;
            len = len - half - 1;
        }
    }
    return first;
}

public Pair<I, I> EqualRange<I, T>(I first, I last, const T& value)
    where I is ForwardIterator and TotallyOrdered<T, I.ValueType>
{
    int len = Distance(first, last);
    while (len > 0)
    {
        int half = len >> 1;
        I middle = Next(first, half);
        if (*middle < value)
        {
            first = middle;
            ++first;
            len = len - half - 1;
        }
        else if (value < *middle)
        {
            len = half;
        }
        else
        {
            I left = LowerBound(first, middle, value);
            I end = Next(first, len);
            ++middle;
            I right = UpperBound(middle, end, value);
            return Pair<I, I>(left, right);
        }
    }
    return Pair<I, I>(first, first);
}

public Pair<I, I> EqualRange<I, T, R>(I first, I last, const T& value
    , R r) where I is ForwardIterator and T is I.ValueType and R is
    Relation and R.Domain is I.ValueType
{
    int len = Distance(first, last);
    while (len > 0)
    {
        int half = len >> 1;
        I middle = Next(first, half);
        if (r(*middle, value))

```

```

        {
            first = middle;
            ++first;
            len = len - half - 1;
        }
        else if (r(value, *middle))
        {
            len = half;
        }
        else
        {
            I left = LowerBound(first, middle, value, r);
            I end = Next(first, len);
            ++middle;
            I right = UpperBound(middle, end, value, r);
            return Pair<I, I>(left, right);
        }
    }
    return Pair<I, I>(first, first);
}

public nothrow I Find<I, T>(I begin, I end, const T& value) where I
    is InputIterator and T is Semiregular and EqualityComparable<T, I.
    ValueType>
{
    while (begin != end)
    {
        if (*begin == value)
        {
            return begin;
        }
        ++begin;
    }
    return end;
}

public nothrow I Find<I, P>(I begin, I end, P p) where I is
    InputIterator and P is UnaryPredicate and P.ArgumentType is I.
    ValueType
{
    while (begin != end)
    {
        if (p(*begin))
        {
            return begin;
        }
        ++begin;
    }
    return end;
}

public nothrow int Count<I, T>(I begin, I end, const T& value) where
    I is InputIterator and T is Semiregular and EqualityComparable<T,

```

```

    I.ValueType>
{
    int count = 0;
    while (begin != end)
    {
        if (*begin == value)
        {
            ++count;
        }
        ++begin;
    }
    return count;
}

public nothrow int Count<I, P>(I begin, I end, P p) where I is
    InputIterator and P is UnaryPredicate and P.ArgumentType is I.
    ValueType
{
    int count = 0;
    while (begin != end)
    {
        if (p(*begin))
        {
            ++count;
        }
        ++begin;
    }
    return count;
}

public nothrow O RemoveCopy<I, O, P>(I begin, I end, O result, P p)
where I is InputIterator and O is OutputIterator and O.ValueType
is I.ValueType and P is UnaryPredicate and P.ArgumentType is I.
    ValueType
{
    while (begin != end)
    {
        if (!p(*begin))
        {
            *result = *begin;
            ++result;
        }
        ++begin;
    }
    return result;
}

public nothrow I Remove<I, P>(I begin, I end, P p) where I is
    ForwardIterator and P is UnaryPredicate and P.ArgumentType is I.
    ValueType
{
    begin = Find(begin, end, p);
    if (begin == end)

```



```

    {
        return begin;
    }
    else
    {
        I i = begin;
        ++i;
        return RemoveCopy(i, end, begin, p);
    }
}

public nothrow O RemoveCopy<I, O, T>(I begin, I end, O result, const
T& value)
    where T is Semiregular and I is InputIterator and O is
        OutputIterator and O.ValueType is I.ValueType and
        EqualityComparable<T, I.ValueType>
{
    while (begin != end)
    {
        if (*begin != value)
        {
            *result = *begin;
            ++result;
        }
        ++begin;
    }
    return result;
}

public nothrow I Remove<I, T>(I begin, I end, const T& value) where I
is ForwardIterator and T is Semiregular and EqualityComparable<T,
I.ValueType>
{
    begin = Find(begin, end, value);
    if (begin == end)
    {
        return begin;
    }
    else
    {
        I i = begin;
        ++i;
        return RemoveCopy(i, end, begin, value);
    }
}

public nothrow void Fill<I, T>(I begin, I end, const T& value) where
    T is Semiregular and I is ForwardIterator and I.ValueType is T
{
    while (begin != end)
    {
        *begin = value;
        ++begin;
    }
}

```

```

    }
}

public nothrow T Accumulate<I, T, Op>(I begin, I end, T init, Op op)
    where I is InputIterator and T is Semiregular and Op is
        BinaryOperation and Op.FirstArgumentType is T and Op.
        SecondArgumentType is I.ValueType
{
    while (begin != end)
    {
        init = op(init, *begin);
        ++begin;
    }
    return init;
}

public F ForEach<I, F>(I begin, I end, F f) where I is InputIterator
    and F is UnaryFunction and F.ArgumentType is I.ValueType
{
    while (begin != end)
    {
        f(*begin);
        ++begin;
    }
    return f;
}

public O Transform<I, O, F>(I begin, I end, O to, F fun)
    where I is InputIterator and O is OutputIterator and F is
        UnaryFunction and F.ArgumentType is I.ValueType and
        CopyAssignable<O.ValueType, F.ResultType>
{
    while (begin != end)
    {
        *to = fun(*begin);
        ++begin;
        ++to;
    }
    return to;
}

public O Transform<I1, I2, O, F>(I1 begin1, I1 end1, I2 begin2, O to,
    F fun)
    where I1 is InputIterator and I2 is InputIterator and O is
        OutputIterator and F is BinaryFunction and F.FirstArgumentType
        is I1.ValueType and F.SecondArgumentType is I2.ValueType and
        CopyAssignable<O.ValueType, F.ResultType>
{
    while (begin1 != end1)
    {
        *to = fun(*begin1, *begin2);
        ++begin1;
        ++begin2;
    }
}

```

```

        ++to;
    }
    return to;
}

public nothrow inline const T& Select_0_2<T, R>(const T& a, const T&
    b, R r) where T is Semiregular and R is Relation and R.Domain is T
{
    if (r(b, a)) return b;
    return a;
}

public nothrow inline const T& Select_1_2<T, R>(const T& a, const T&
    b, R r) where T is Semiregular and R is Relation and R.Domain is T
{
    if (r(b, a)) return a;
    return b;
}

public nothrow inline const T& Select_0_3<T, R>(const T& a, const T&
    b, const T& c, R r) where T is Semiregular and R is Relation and R
    .Domain is T
{
    return Select_0_2(Select_0_2(a, b, r), c, r);
}

public nothrow inline const T& Select_2_3<T, R>(const T& a, const T&
    b, const T& c, R r) where T is Semiregular and R is Relation and R
    .Domain is T
{
    return Select_1_2(Select_1_2(a, b, r), c, r);
}

public nothrow inline const T& Select_1_3_ab<T, R>(const T& a, const
    T& b, const T& c, R r) where T is Semiregular and R is Relation
    and R.Domain is T
{
    if (!r(c, b)) return b;
    return Select_1_2(a, c, r);
}

public nothrow inline const T& Select_1_3<T, R>(const T& a, const T&
    b, const T& c, R r) where T is Semiregular and R is Relation and R
    .Domain is T
{
    if (r(b, a)) return Select_1_3_ab(b, a, c, r);
    return Select_1_3_ab(a, b, c, r);
}

public nothrow const T& Median<T, R>(const T& a, const T& b, const T&
    c, R r) where T is Semiregular and R is Relation and R.Domain is
    T
{

```

```

    return Select_1_3(a, b, c, r);
}

public nothrow const T& Median<T>(const T& a, const T& b, const T& c)
    where T is TotallyOrdered
{
    return Median(a, b, c, Less<T>());
}

public nothrow I UnguardedPartition<I, T, R>(I begin, I end, const T&
    pivot, R r) where I is RandomAccessIterator and T is I.ValueType
    and R is Relation and R.Domain is I.ValueType
{
    while (true)
    {
        while (r(*begin, pivot))
        {
            ++begin;
        }
        --end;
        while (r(pivot, *end))
        {
            --end;
        }
        if (begin >= end)
        {
            return begin;
        }
        Swap(*begin, *end);
        ++begin;
    }
    // dummy return to keep compiler happy...
    return begin;
}

public nothrow void UnguardedLinearInsert<I, T, R>(I last, const T&
    val, R r) where I is RandomAccessIterator and T is I.ValueType and
    R is Relation and R.Domain is I.ValueType
{
    I next = last;
    --next;
    while (r(val, *next))
    {
        *last = *next;
        last = next;
        --next;
    }
    *last = val;
}

public void LinearInsert<I, R>(I first, I last, R r) where I is
    RandomAccessIterator and R is Relation and R.Domain is I.ValueType
{

```

```

    I.ValueType val = *last;
    if (r(val, *first))
    {
        CopyBackward(first, last, last + 1);
        *first = val;
    }
    else
    {
        UnguardedLinearInsert(last, val, r);
    }
}

public void InsertionSort<I, R>(I begin, I end, R r) where I is
    RandomAccessIterator and R is Relation and R.Domain is I.ValueType
{
    if (begin == end)
    {
        return;
    }
    for (I i = begin + 1; i != end; ++i)
    {
        LinearInsert(begin, i, r);
    }
}

public inline void InsertionSort<I>(I begin, I end) where I is
    RandomAccessIterator and I.ValueType is TotallyOrdered
{
    InsertionSort(begin, end, Less<I.ValueType>());
}

public const int InsertionSortThreshold = 16;

public void PartialQuickSort<I, R>(I begin, I end, R r) where I is
    RandomAccessIterator and R is Relation and R.Domain is I.ValueType
{
    while (end - begin > InsertionSortThreshold)
    {
        I.ValueType pivot = Median(*begin, *(begin + (end - begin) /
            2), *(end - 1), r);
        I cut = UnguardedPartition(begin, end, pivot, r);
        PartialQuickSort(cut, end, r);
        end = cut;
    }
}

public void Sort<I, R>(I begin, I end, R r) where I is
    RandomAccessIterator and R is Relation and R.Domain is I.ValueType
{
    if (begin != end)
    {
        PartialQuickSort(begin, end, r);
        InsertionSort(begin, end, r);
    }
}

```

```

    }
}

public inline void Sort<I>(I begin, I end) where I is
    RandomAccessIterator and I.ValueType is TotallyOrdered
{
    Sort(begin, end, Less<I.ValueType>());
}

public inline void Sort<C, R>(C& c, R r) where C is
    RandomAccessContainer and R is Relation and R.Domain is C.Iterator
    .ValueType
{
    Sort(c.Begin(), c.End(), r);
}

public inline void Sort<C>(C& c) where C is RandomAccessContainer and
    C.Iterator.ValueType is TotallyOrdered
{
    Sort(c.Begin(), c.End());
}

public void Sort<C>(C& c) where C is ForwardContainer and C.Iterator.
    ValueType is TotallyOrdered
{
    List<C.ValueType> list;
    Copy(c.CBegin(), c.CEnd(), BackInserter(list));
    Sort(list);
    Copy(list.CBegin(), list.CEnd(), c.Begin());
}

public void Sort<C, R>(C& c, R r) where C is ForwardContainer and R
    is Relation and R.Domain is C.Iterator.ValueType
{
    List<C.ValueType> list;
    Copy(c.CBegin(), c.CEnd(), BackInserter(list));
    Sort(list, r);
    Copy(list.CBegin(), list.CEnd(), c.Begin());
}

public nothrow bool Equal<I1, I2, R>(I1 first1, I1 last1, I2 first2,
    I2 last2, R r) where I1 is InputIterator and I2 is InputIterator
    and Relation<R, I1.ValueType, I2.ValueType>
{
    while (first1 != last1 && first2 != last2)
    {
        if (!r(*first1, *first2))
        {
            return false;
        }
        ++first1;
        ++first2;
    }
}

```

```

    return first1 == last1 && first2 == last2;
}

public nothrow inline bool Equal<I1, I2>(I1 first1, I1 last1, I2
first2, I2 last2) where I1 is InputIterator and I2 is
InputIterator and EqualityComparable<I1.ValueType, I2.ValueType>
{
    return Equal(first1, last1, first2, last2, EqualTo2<I1.ValueType,
I2.ValueType>());
}

public nothrow bool LexicographicalCompare<I1, I2, R>(I1 first1, I1
last1, I2 first2, I2 last2, R r)
    where I1 is InputIterator and I2 is InputIterator and Same<I1.
ValueType, I2.ValueType> and Relation<R, I1.ValueType, I2.
ValueType> and Relation<R, I2.ValueType, I1.ValueType>
{
    while (first1 != last1 && first2 != last2)
    {
        if (r(*first1, *first2))
        {
            return true;
        }
        if (r(*first2, *first1))
        {
            return false;
        }
        ++first1;
        ++first2;
    }
    return first1 == last1 && first2 != last2;
}

public nothrow inline bool LexicographicalCompare<I1, I2>(I1 first1,
I1 last1, I2 first2, I2 last2) where I1 is InputIterator and I2 is
InputIterator and LessThanComparable<I1.ValueType, I2.ValueType>
{
    return LexicographicalCompare(first1, last1, first2, last2, Less2
<I1.ValueType, I2.ValueType>());
}

public nothrow I MinElement<I>(I first, I last) where I is
ForwardIterator and I.ValueType is TotallyOrdered
{
    if (first == last)
    {
        return first;
    }
    I minElementPos = first;
    ++first;
    while (first != last)
    {
        if (*first < *minElementPos)

```

```

        {
            minElementPos = first;
        }
        ++first;
    }
    return minElementPos;
}

public nothrow I MinElement<I, R>(I first, I last, R r) where I is
    ForwardIterator and R is Relation and R.Domain is I.ValueType
{
    if (first == last)
    {
        return first;
    }
    I minElementPos = first;
    ++first;
    while (first != last)
    {
        if (r(*first, *minElementPos))
        {
            minElementPos = first;
        }
        ++first;
    }
    return minElementPos;
}

public nothrow I MaxElement<I>(I first, I last) where I is
    ForwardIterator and I.ValueType is TotallyOrdered
{
    if (first == last)
    {
        return first;
    }
    I maxElementPos = first;
    ++first;
    while (first != last)
    {
        if (*maxElementPos < *first)
        {
            maxElementPos = first;
        }
        ++first;
    }
    return maxElementPos;
}

public nothrow I MaxElement<I, R>(I first, I last, R r) where I is
    ForwardIterator and R is Relation and R.Domain is I.ValueType
{
    if (first == last)
    {

```



```

        return first;
    }
    I maxElementPos = first;
    ++first;
    while (first != last)
    {
        if (r(*maxElementPos, *first))
        {
            maxElementPos = first;
        }
        ++first;
    }
    return maxElementPos;
}

public nothrow inline T Abs<T>(const T& x) where T is
    OrderedAdditiveGroup
{
    if (x < T(0))
    {
        return -x;
    }
    return x;
}

// naive implementation...
public nothrow U Factorial<U>(U n) where U is UnsignedInteger
{
    U f = 1u;
    for (U u = 2u; u <= n; ++u)
    {
        f = f * u;
    }
    return f;
}

public nothrow T Gcd<T>(T a, T b) where T is EuclideanSemiring
{
    while (true)
    {
        if (b == T(0)) return a;
        a = a % b;
        if (a == T(0)) return b;
        b = b % a;
    }
}

public nothrow bool NextPermutation<I>(I begin, I end) where I is
    BidirectionalIterator and I.ValueType is LessThanComparable
{
    if (begin == end)
    {
        return false;
    }

```

```

    }
    I i = begin;
    ++i;
    if (i == end)
    {
        return false;
    }
    i = end;
    --i;
    while (true)
    {
        I ii = i;
        --i;
        if (*i < *ii)
        {
            I j = end;
            --j;
            while (*i >= *j)
            {
                --j;
            }
            Swap(*i, *j);
            Reverse(ii, end);
            return true;
        }
        if (i == begin)
        {
            Reverse(begin, end);
            return false;
        }
    }
}

public nothrow bool NextPermutation<I, R>(I begin, I end, R r) where
    I is BidirectionalIterator and R is Relation and R.Domain is I.
    ValueType
{
    if (begin == end)
    {
        return false;
    }
    I i = begin;
    ++i;
    if (i == end)
    {
        return false;
    }
    i = end;
    --i;
    while (true)
    {
        I ii = i;
        --i;

```

```

        if (r(*i, *ii))
        {
            I j = end;
            --j;
            while (!r(*i, *j))
            {
                --j;
            }
            Swap(*i, *j);
            Reverse(ii, end);
            return true;
        }
        if (i == begin)
        {
            Reverse(begin, end);
            return false;
        }
    }
}

public nothrow bool PrevPermutation<I>(I begin, I end) where I is
BidirectionalIterator and I.ValueType is LessThanComparable
{
    if (begin == end)
    {
        return false;
    }
    I i = begin;
    ++i;
    if (i == end)
    {
        return false;
    }
    i = end;
    --i;
    while (true)
    {
        I ii = i;
        --i;
        if (*ii < *i)
        {
            I j = end;
            --j;
            while (*j >= *i)
            {
                --j;
            }
            Swap(*i, *j);
            Reverse(ii, end);
            return true;
        }
        if (i == begin)
        {

```

```

        Reverse(begin, end);
        return false;
    }
}

public nothrow bool PrevPermutation<I, R>(I begin, I end, R r) where
    I is BidirectionalIterator and R is Relation and R.Domain is I.
    ValueType
{
    if (begin == end)
    {
        return false;
    }
    I i = begin;
    ++i;
    if (i == end)
    {
        return false;
    }
    i = end;
    --i;
    while (true)
    {
        I ii = i;
        --i;
        if (r(*ii, *i))
        {
            I j = end;
            --j;
            while (!r(*j, *i))
            {
                --j;
            }
            Swap(*i, *j);
            Reverse(ii, end);
            return true;
        }
        if (i == begin)
        {
            Reverse(begin, end);
            return false;
        }
    }
}

public nothrow inline uint RandomNumber(uint n)
{
    return Rand() % n;
}

public nothrow void RandomShuffle<I>(I begin, I end) where I is
    RandomAccessIterator

```

```
{  
    if (begin == end) return;  
    for (I i = begin + 1; i != end; ++i)  
    {  
        int d = (i - begin) + 1;  
        int r = cast<int>(RandomNumber(cast<uint>(d)));  
        I j = begin + r;  
        Swap(*i, *j);  
    }  
}
```