```
    public enum AverageSetSize
    {
        ThirtyTwo = 0,
        TweentyOne = 1
    }

    public class ExtendedHashSet<T> : HashSet<T>
    {
        private ulong _productA = 1;
        private ulong _productB = 1;
        private ulong _productC = 1;
        private AverageSetSize _averageSetSize;

        public ExtendedHashSet(AverageSetSize averageSetSize = AverageSetSize.ThirtyTwo)
: base() { Initialize(averageSetSize); }
        public ExtendedHashSet(IEqualityComparer<T> comparer, AverageSetSize
averageSetSize = AverageSetSize.ThirtyTwo) : base(comparer) { Initialize(averageSetSize);
}
        public ExtendedHashSet(IEnumerable<T> collection, AverageSetSize averageSetSize =
AverageSetSize.ThirtyTwo) : base() { Initialize(averageSetSize, collection); }
        public ExtendedHashSet(IEnumerable<T> collection, IEqualityComparer<T> comparer,
AverageSetSize averageSetSize = AverageSetSize.ThirtyTwo) : base(comparer) {
Initialize(averageSetSize, collection); }

        private void Initialize(AverageSetSize averageSetSize, IEnumerable<T> collection
= null)
        {
            _averageSetSize = averageSetSize;
            if (collection != null)
            {
                foreach(T item in collection)
                {
                    Add(item);
                }
            }
        }

        public new bool Add(T item)
        {
            if (base.Add(item) == true)
            {
                if (_productA > 0)
                {
                    int hashCode = item.GetHashCode();
                    uint a = (uint)(hashCode & (_averageSetSize ==
AverageSetSize.ThirtyTwo ? 7 : 15)) + 1;
                    uint b = (uint)((hashCode >> (_averageSetSize ==
AverageSetSize.ThirtyTwo ? 3 : 4)) & (_averageSetSize == AverageSetSize.ThirtyTwo ? 7 :
15)) + 1;
                    uint c = (uint)((hashCode >> (_averageSetSize ==
AverageSetSize.ThirtyTwo ? 6 : 8)) & (_averageSetSize == AverageSetSize.ThirtyTwo ? 7 :
15)) + 1;

                    try
                   {
                        checked
                        {
                            _productA *= a;
```

```
                        _productB *= b;
                        _productC *= c;
                    }
                }
                catch
                {
                    //if overflow happens
                    _productA = 0;
                    _productB = 0;
                    _productC = 0;
                }
            }
            return true;
        }
        return false;
    }

    public new bool Remove(T item)
    {
        if (base.Remove(item) == true)
        {
            int hashCode = item.GetHashCode();
            uint a = (uint)(hashCode & (_averageSetSize == AverageSetSize.ThirtyTwo ?
7 : 15)) + 1;
            uint b = (uint)((hashCode >> (_averageSetSize == AverageSetSize.ThirtyTwo
? 3 : 4)) & (_averageSetSize == AverageSetSize.ThirtyTwo ? 7 : 15)) + 1;
            uint c = (uint)((hashCode >> (_averageSetSize == AverageSetSize.ThirtyTwo
? 6 : 8)) & (_averageSetSize == AverageSetSize.ThirtyTwo ? 7 : 15)) + 1;

            _productA /= a;
            _productB /= b;
            _productC /= c;

            return true;
        }
        return false;
    }

    public new void ExceptWith(IEnumerable<T> other)
    {
        foreach(T item in other)
        {
            Remove(item);
        }
    }

    public new int RemoveWhere(Predicate<T> match)
    {
        int count = 0;
        LinkedList<T> removedItems = new LinkedList<T>();
        foreach (T item in this)
        {
            if(match(item) == true)
            {
                removedItems.AddLast(item);
            }
        }
        foreach(T item in removedItems)
```

```
    {
        if(Remove(item) == true)
        {
            count++;
        }
    }
    return count;
}

public new void IntersectWith(IEnumerable<T> other)
{
    RemoveWhere((T item) =>
    {
        foreach (T otherItem in other)
        {
            if (Comparer.Equals(item, otherItem) == true)
            {
                return true;
            }
        }
        return false;
    });
}

public new void SymmetricExceptWith(IEnumerable<T> other)
{
    foreach(T item in other)
    {
        if(Contains(item) == true)
        {
            Remove(item);
        }
        else
        {
            Add(item);
        }
    }
}

public new void UnionWith(IEnumerable<T> other)
{
    foreach (T item in other)
    {
        if (Contains(item) == false)
        {
            Add(item);
        }
    }
}

public new void Clear()
{
    _productA = 1;
    _productB = 1;
    _productC = 1;
    base.Clear();
}
```

```
        public new bool SetEquals(IEnumerable<T> other)
        {
            if (other == this)
            {
                return true;
            }
            if (other is ExtendedHashSet<T>)
            {
                ExtendedHashSet<T> set = (ExtendedHashSet<T>)other;
                if (_productA == set._productA && _productB == set._productB && _productC
== set._productC)
                {
                    return base.SetEquals(other);
                }
            }
            return base.SetEquals(other);
        }

        public bool IsSubsetOf(ExtendedHashSet<T> set)
        {
            //if "this" has a greater size than "set" or "this" overflows and "set" does
not, can't be a subset
            if (Count > set.Count || (_productA == 0 && set._productA != 0))
            {
                return false;
            }

            //if "set" overflows, have to do a full check
            //check if "this" products divide "set" products, if they do, we need to do a
full check
            if (set._productA == 0 || (set._productA % _productA == 0 && set._productB %
_productB == 0 && set._productC % _productC == 0))
            {
                return base.IsSubsetOf(set);
            }

            return false;
        }

        public bool IsSupersetOf(ExtendedHashSet<T> set)
        {
            return set.IsSubsetOf(this);
        }

        public bool IsProperSubsetOf(ExtendedHashSet<T> set)
        {
            if (Count == set.Count)
            {
                return false;
            }
            return IsSubsetOf(set);
        }

        public bool IsProperSupersetOf(ExtendedHashSet<T> set)
        {
            return set.IsProperSubsetOf(this);
        }
    }
```