```
public static class InterleavingApplyClass
        public static IEnumerable<TRes> InterleavingApply<T, TRes>
             (IEnumerable<T> s1, IEnumerable<T> s2, Func<T,TRes> f)
        {
            IEnumerable<TRes> InterleavingApply_Aux()
            {
                TRes aux(T elem)
                {
                    try
                    {
                        return f(elem);
                    }
                    catch(Exception e)
                    {
                        throw new FunctionApplicationException("errore nella
                         funzione", e);
                    }
                }

                using var it1 = s1.GetEnumerator();
                using var it2 = s2.GetEnumerator();
                bool check1, check2;
                while (true)
                {
                    check1 = it1.MoveNext();
                    check2 = it2.MoveNext();
                    if (check1 == false && check2 == false) break;
                    if(check1 == false && check2)
                         throw new DifferentLenghtException(false);
                    if (check1 && check2 == false)
                         throw new DifferentLenghtException();
                    yield return aux(it1.Current);
                    yield return aux(it2.Current);

                }

            }
            return InterleavingApply_Aux();
        }
    }

    public class InterleavingApplyTest
    {
        [Test]
        public void NormalBehaviour() =>
            Assert.That(
             InterleavingApplyClass
             .InterleavingApply(new[] { 8, 11, 35 },new[] { 100, 34, 23 },
              (elem) => elem / 7),
                Is.EqualTo(new[] {1,14,1,4,5,3}));

        [TestCase(new[] {'a','b','c'}, new[] {'@','#','§'})]
        [TestCase(new[] { 'a', 'b', 'c','2','7' }, new[] { '@', '#', '§' })]
        [TestCase(new[] { 'a', 'b', 'c' }, new[] { '@', '#', '§','2' })]
        public void DifferentLenght(IEnumerable<char> s1, IEnumerable<char> s2)
        {
            if (s1.ToList().Count == s2.ToList().Count) Assert.Inconclusive();
            Func<char, bool> isDigit = c => c >= '0' && c <= '9';
            Assert.That(() => InterleavingApplyClass
             .InterleavingApply(s1, s2, isDigit).ToArray(),
             Throws.TypeOf<DifferentLenghtException>());
        }
```

```
    [Test]
    public void InfiniteSequenceTest()
    {
        IEnumerable<int> InfiniteSeq(int[] v)
        {
            int i = 0;
            while (true)
            {
                yield return v[i++ % v.Length];
            }
        }

        IEnumerable<bool> InfiniteBoolSeq()
        {
            while (true)
            {
                yield return false;
                yield return true;
            }
        }

        Func<int, bool> isEven = n => n % 2 == 0;
        Assert.That(
            InterleavingApplyClass.InterleavingApply(
                InfiniteSeq(new[] { 7, 5, 17 }).Take(100),
                InfiniteSeq(new[] { 2, 18, 42, 128, 512 }).Take(100),
                isEven).Take(100)
                ,Is.EqualTo(InfiniteBoolSeq().Take(100)));
    }
}
```