1. Partition and Related Functions

Reverse, rotate and random shuffle are the most important examples of index-based permutations, that is, permutations that rearrange a sequence according to the original position of the elements without any consideration for their values. Now we are going to study a different class of permutation algorithms, predicate-based permutations. The positions into which these algorithms move elements in a sequence depend primarily on whether they satisfy a given condition, not only on their original position.

Most of the algorithms in this section are based on the notion of *partition*: dividing elements in a range into *good* and *bad* according to a predicate. One of the tasks of the present chapter is to show the richness of the algorithmic space surrounding partition. We will find use for many of the functions that we studied before, such as find, reduce and rotate. We will discover many techniques and interfaces that will serve us well in the following chapters.

Definitions

1. A range is *partitioned* according to a given predicate if every element in the range that satisfies the predicate precedes every element that does not satisfy the predicate.

2. An iterator `m` into a partitioned range `[f, l)` is called a *partition point* if every element in the range `[f, m)` satisfies the predicate and no element in the range `[m, l)` satisfies it.

For example, if `G` stands for *good* (satisfying), `B` for *bad*(unsatisfying) elements then the following range `[f, l)` is partitioned and `m` is its partition point:

```
GGGGGBBB
^    ^  ^
f    m  l
```

Notice, that as we have seen in cases of other algorithms, partition requires N + 1 different values of iterators to describe all possible partition points of a sequence of N elements. Indeed, if we have N elements in a sequence the number of good elements in it varies between 0 and N, having, therefore, N + 1 distinct values.

We can check if a range is partitioned using the following function:

```
template <typename I,  // I models Input Iterator
          typename P>  // P models Unary Predicate
bool is_partitioned(I f, I l, P p)
```

```
{
     return l == find_if(find_if_not(f, l, p), l, p);
}
```

The function checks that there are no good elements that follow a bad element.

If we know the partition point `m` we can verify the partitioning with:

```
template <typename I,  // I models Input Iterator
          typename P>  // P models Unary Predicate
bool is_partitioned(I f, I m, I l, P p)
{
     return all(f, m, p) && none(m, l, p);
}
```

If a range is partitioned according to some predicate we can easily find its partition point by calling:

```
find_if_not(f, l, p)
```

Later in this section we shall see that it is often possible to find the partition point much faster.

Quiz: How is it possible to find the partition point faster?

There could be many different permutations of a range that give us a partitioned range. If we have a range with `U` good elements and `V` bad elements the number of different partitioned permutations is equal to `U!V!`.

Problem: How large must a range be to have different partitioned permutations irrespective of the number of good and bad elements in it?

Forward Partition Algorithm (Lomuto)

In order to partition a range `[f, l)` in place we start with an inductive assumption:

Let us assume that we managed to partition a range up to some point `n` and the present partition point is `m`. We can illustrate the current state with the picture:

```
     GGGGGGGGBBBBBBB??????
     ^       ^      ^     ^
     f       m      n     l
```

where `G` stands for “good” (satisfying), `B` for “bad”(unsatisfying) and `?` stands for “untested”. Then we know that:

```
      assert(all(f, m, p) && none(m, n, p)); // invariant
```

We do not know anything about the value at `n`. Before we check if it satisfies the predicate we need to assure that we have not reached the end of the range. But if somehow we did, we are done. Indeed if `n` is equal to `l` then our loop invariant becomes equivalent to the second version of `is_partitioned` which happens to be the postcondition of the function that we are trying to implement. It is evident that we should return the partition point. Indeed, we have it available and it might be - and in reality almost invariably is - useful to the caller.

Now we have the innermost part of our program:

```
      assert(all(f, m, p) && none(m, n, p)); // invariant
      if (n == l) return m;
```

Since we have not reached the end of the range we can test the next element.

If the element to which `n` points does not satisfy the predicate, we can simply advance `n` and our invariant still holds. Otherwise, we swap a good element pointed to by `n` with a (usually) bad element pointed to by `m` and we can advance both m and n with our invariant holding.

Quiz: Can there ever be the case that `m` points to a good element? Would the invariant still hold if it does?

Since we know that eventually `n` will reach `l` our program is almost done:

```
while (true) {
      assert(all(f, m, p) && none(m, n, p)); // invariant
      if (n == l)
            return m;
      if (p(*n)) {
            swap(*n, *m);
            ++m;
      }
      ++n;
}
```

We observe that for any range `[f, l)` we can find our inductive base by starting with both `m` and `n` being equal to `f`. Or, stating it differently, it is really easy to partition an empty range and find its partition point. And since now we have both the starting point for our induction and the inductive step, we obtain:

```
template <typename I,  // I models Forward Iterator
          typename P>  // P models Unary Predicate
I partition_forward_unoptimized(I f, I l, P p)
{
     I m = f;
     I n = f;
     while (n != l) {
          if (p(*n)) {
               swap(*n, *m);
               ++m;
          }
          ++n;
     }
     assert(is_partitioned(f, m, l, p));
     return m;
}
```

Remark: It is interesting that this excellent algorithm is not in the C++ standard which requires bidirectional iterators for partition. I had known, implemented, and taught this algorithm for quite some time – since I first read about it in Bentley's column in CACM in the mid-eighties. But my original STL proposal does, somehow, specify bidirectional iterators for both `partition` and `stable_partition`. Both of them were corrected in SGI STL, but most vendors are still behind. This little thing has been bothering me for over 10 years now; the most bothersome part being the fact of omission. How did it happen? I suspect that the explanation is quite simple: while in the early 90ties I already understood the idea of reducing every algorithms to its minimal requirements, and I also knew that the same operation could be implemented using better algorithms when we know more about the data to which they are applied, I was not yet fully aware of the need to provide an algorithm for the weakest case, if such an algorithm is available. It took several more years to understand the importance of "filling the algorithmic space."

How many operations does the algorithm perform? The number of the applications of the predicate is exactly equal to the length of the range. And that is, indeed, the minimal number possible.

Assuming that N is the length of a range solve the following problems:

Problem: Prove that it is not possible to find the partition point with fewer than N predicate applications.

Problem: Prove that if it is not required to return a partition point then it is possible to partition a non-empty range with fewer than N predicate applications. [Jon Brandt]

Problem: Prove that even without returning a partition point it is not possible to partition a range with fewer than N-1 predicate applications.

While the algorithm is optimal in terms of the number of application of the predicate it clearly does more swaps than necessary. Indeed, it does one swap for every good element in the sequence. But it is absolutely unnecessary to do it when there is no preceding bad element. We can, therefore, produce an optimized version of the algorithm that skips over all the good elements in the beginning of the range. We can also optimize away one of the iterator variables:

```
template <typename I,  // I models Forward Iterator
          typename P>  // P models Unary Predicate
I partition_forward_1(I f, I l, P p)
{
     f = find_if_not(f, l, p);
     if (f == l) return f;
     I n = f;
     while (++n != l)
          if (p(*n)) {
               swap(*n, *f);
               ++f;
          }
     return f;
}
```

While it seems to be a worthwhile optimization, in reality it is not very useful since the average number of good elements in front of the first bad element is going to be very small. We are, therefore, saving just a constant number of operations in a linear algorithm, which, in general, is not a very useful optimization. The main reason for doing it is esthetic: the optimized version is not going to do any swaps if a range is already partitioned, which is a “nice” (but not practically useful) property.

Problem: What is the average number of good elements in front of the first bad element?

Now the number of swaps is going to be equal to the number of the good elements that appear in the range after the first bad element. While it is “optimal” for this algorithm, it is clearly excessive. For example, if we have a sequence of one bad element followed by four good elements:

```
BGGGG
```

our program is going to perform four swaps, while a partitioned sequence can be obtained with a single swap of the first and the fifth elements. It is easy to see that on the average there will be approximately N/2 good elements after a bad element and, therefore, on the average the algorithm will do N/2 swaps.

What is the minimal number of swaps that are needed for partition? Well, as a matter of fact the question is not particularly interesting. In terms of minimal number of moving operations we should ask about what is the minimal number of moves that are needed to partition a given range. The answer is simple: if we have a range with U good elements and V bad elements and there are K bad elements amongst the first U elements of the range, then we need 2K+1 moves to partition the range (assuming, of course, that K is not equal to 0). Indeed, K bad elements are out of place and so there are K good elements that are originally positioned outside of their final destination. To move these 2K elements we need at least 2K moves and we need one extra location where we need to save one of the elements to enable us to initiate the sequence of moves.

Problem: Design a partition algorithm that does 2K+1 moves. You do not have to assume that iterators are forward iterators. [Solution can be found later in this section]

What is the number of iterator operations performed by `partition_forward`? It is clear that we need to do N iterator comparisons to watch for the end. Our present implementation will do an extra one, since it will compare an iterator returned by find which would not have been necessary if we decided to hand-inline find and obtained the following code sequence:

```
template <typename I,  // I models Forward Iterator
          typename P>  // P models Unary Predicate
I partition_forward(I f, I l, P p)
{
     while (true) {
          if (f == l) return f;
          if (!p(*f)) break;
          ++f;
     }
     I n = f;
     while (++n != l)
          if (p(*n)) {
               swap(*n, *f);
               ++f;
          }
     return f;
}
```

In this context the optimization is not particularly useful since a single extra comparison does not really effect the performance (a small constant added to a linear function), but we encounter the same transformation in the next algorithm where the extra comparison appears in the inner loop. The transformation starts with the loop of `find_if_not`:

```
while (f != l && p(*f)) ++f;
```

and provides two different exits depending on which part of the conjunction holds. The total number of iterator increments is equal to N + W where W is the number of good elements that follow the first bad element. As we remarked before, on the average it is going to be approximately N + U – 2 where U is the number of good elements in the range.

Remark: We do not know a partition algorithm that is more effective for forward iterators than the one we just described. We believe that in some fundamental sense it is optimal, but we do not even know how to state the problem. We typically analyze the algorithmic performance by counting one kind of operation. In reality we are dealing with several different operations. For partition we need predicate application and move (both of which depend on the type of elements) and iterator increment and equality (both of which depend on the iterator type). We have a general feeling that element operations are potentially costlier than iterator operations, since elements could be large while iterators are small. Such vague considerations usually allow us to produce algorithms that are satisfactory in practice, but there is something unsatisfying about it. It is possible that one can come up with axioms on the complexity measures of different operations that will allow us to prove optimality of certain algorithms. So far, we failed either to design such axioms or to interest others in solving such problems.

Bidirectional Partition Algorithm (Hoare)

While, as we shall see later, it is possible to implement a partition algorithm with a minimal number of moves, in practice it is usually sufficient to replace 2K +1 moves with K swaps, namely, discover all the K misplaced bad elements and swap them with K misplaced good elements. Our goal is to assure that every swap puts both the good element and the bad element in their final destination. If we take the rightmost good element and the leftmost bad element we can be sure that if they are out of place we can put both in acceptable positions by swapping them. Indeed, we know that all the elements to the left of the leftmost bad element have to be good and are in their final destination; and similarly for the rightmost good element. So if they are out of place – the leftmost bad element is before the rightmost good element, then swapping them is putting both into acceptable locations. Finding the rightmost good element efficiently requires that we move from the right and that requires bidirectional iterators.

The idea of the algorithm can be illustrated by the following picture:

```
GGGGGGGGB??????GBBBBB
^       ^      ^     ^
f0      f      l     l0
```

Interchanging the elements pointed to by `f` and `l` will put them in the correct subranges: the bad element to the right of the partition point and the good element to the left of it. It is worthwhile to observe that the partition point is located somewhere in the range `[f, l)`.

We can start our implementation by first finding the new `f`, then finding the new `l` and then swapping them or returning whichever one is appropriate:

```
// use find_if_not to find the first bad element
// use find_backward_if to find the last good element
// check if the iterators crossed and return
// swap bad and good elements
```

Before we try to figure out how it works let us have a detour and learn about `find_backward`.

Misplaced section: Find Backward

<<We did not discuss finding backward in our chapter on find. The main reason for that was that our design for its interface might be better understood next to the first example of its use. But we might eventually decide to move it there.>>

It is often important to find elements in a range while traversing it backwards. It seems to be an easy task; just take `find` and replace ++ with `--`:

```
template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I buggy_find_backward_if_1(I f, I l, P p)
{
     while (f != l && !p(*l)) --l;
     return l;
}
```

This, of course, will not work since the first time around we will be dereferencing a past-the-end iterator. We should remember that our ranges are semi-open intervals and the end iterator is not symmetrical with the begin iterator. It seems that we can compensate for it by writing:

```
template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I buggy_find_backward_if_2(I f, I l, P p)
{
     while (f != l && !p(*--l));
     return l;
}
```

The problem now is that we cannot distinguish between finding a good element in the very beginning of the range – but at the end of our search – and not finding a good element at all. We can, of course find out which one is true by re-testing the first element, but it would require an extra test and would not be symmetric with the ordinary `find_if`. It would be terribly nice if we could transform a semi-open range `[f, l)` into a semi-open range `[l, f)`. And we can do it if we just slightly modify our code by incrementing `l` before returning it when we find a good element:

```
template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I find_backward_if(I f, I l, P p)
{
     while (true) {
          if (f == l) return f;
          if (p(*--l)) return ++l;
     }
}
```

We return the first iterator if we do not find a good element; otherwise, we return the successor to the iterator pointing to the first good element from the right. (We assume that ranges grow from left to right.)

End of misplaced section: Find Backward

And now back to our partition. We will carefully write down the asserts:

```
f = find_if_not(f0, l0, p);
     assert(f == l0 || !p(*f) &&
          all(f0, f, p));
l = find_backward_if(f, l0, p);
     assert(f == l ||
          p(*predecessor(l)) &&
          none(l, l0, p);
if (f == l) return f;
     assert(f != l && f != l0 &&
          !p(f) && p(*predecessor(l)) &&
          distance(f, l) > 1);
--l;
swap(*f, *l);
     assert(p(f) && !p(*l));
++f;
     assert(all(f0, f, p) && none(l, l0, p));
```

It is very important that you follow along and assure yourself that all the asserts hold.

Now it is easy to see our program:

```
template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I partition_bidirectional_1(I f, I l, P p)
{
     while (true) {
          f = find_if_not(f, l, p);
          l = find_backward_if(f, l, p);
          if (f == l) return f;
          --l;
          swap(*f, *l);
          ++f;
     }
}
```

The above code looks so elegant, so perfect that it makes us sad that we have to muck it up. But muck it we shall. The present code does several extra operations. As far as the number of swaps goes, it does the promised K of them. However it should be clear that it often does more than the necessary predicate applications.

Quiz: How many extra predicate applications does the algorithm do?

Remark: While one or two extra application of the predicate usually do not matter – and as we shall see soon a few extra application could in reality speed up the algorithm by allowing us to trade a linear number of iterator comparisons for an extra predicate call – sometimes it is important to assure that the algorithm does not do any extra predicate applications. It usually happens when the predicate is not strictly functional and applying the predicate to the same element twice might not yield the same results. The useful example of using partition with such a predicate comes up in an attempt to design an algorithm for randomly shuffling a range with only forward iterator traversal. As we remarked in the section on random shuffle (pages ???) we believe that it is impossible to have a linear time algorithm for random shuffle unless the range provides us with random access iterators. There is, however, an NlogN algorithm that randomly shuffles a range with forward iterators only which is based on using partition with a coin-tossing predicate – a predicate which returns a uniformly random sequence of true and false when applied to any element.

Problem: Implement a function that uses partition on a range to randomly shuffle it [Raymond Lo and Wilson Ho].

Problem: Prove that your implementation of random shuffle does, indeed, produce a uniformly random shuffle [Raymond Lo and Wilson Ho].

In addition to extra predicate applications our `partition_bidirectional_1` function does more than the necessary iterator comparisons. We could patch all these minor problems by inlining our finds and doing the different exit transformation that we first introduced in the previous section:

```
template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I partition_bidirectional_2(I f, I l, P p)
{
     while (true) {
          while (true) {
               if (f == l) return f;
               if (!p(*f)) break;
               ++f;
          }
          while (true) {
               --l;
               if (f == l) return f;
               if (p(*l)) break;
          }
          swap(*f, *l);
          f++;
     }
}
```

Problem: Prove the program correct by carefully writing asserts.

As a matter of fact, we did not muck it up too badly. It still looks very symmetric, very elegant, but as we shall see soon the mucking is not over.

As far as the number of operations goes, the present code does N predicate applications (prove it!) and N+1 iterator comparisons and N+1 iterator increments and decrements. It also – as promised – does K swaps.

Minimizing moves

Sometimes we need to study a subject even if at the end we find out that it has few practical applications. Implementing partition with the minimal number of moves is one such subject. As we have seen earlier in the section, the minimal number of moves necessary for partitioning a range is equal to 2K + 1 where K is the number of bad elements that precede the (eventual) partition point. While we have an algorithm that does K swaps, it does not appear to be optimal since we usually consider a swap to be equivalent to 3 moves and 3K is greater than 2K + 1 for most positive integers. (It is optimal, indeed, when K is 1 and we are going to do a single swap.)

Now let us see how we can produce a version with the minimal number of moves. The idea is quite simple we save the first misplaced element and then move other misplaced elements into the holes formed by the first save and the subsequent moves. When we reach the end, we move the saved element into the last hole. In other words, we re-organize our partition permutation from one with K cycles to one with one cycle.

It should be noted that the result of the algorithm is going to be different from the result of our `partition_bidirectional` which generates a somewhat different permutation.

Now it is fairly straightforward to obtain its implementation:

```
// as usual we use the following macro
#define VALUE_TYPE(I) std::iterator_traits<I >::value_type


template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I partition_bidirectional_minimal_moves (I f, I l, P p)
{
     while (true) {
          if (f == l) return f;
          if (!p (*f)) break;
          ++f;
     } // f points to a bad element
     while (true) {
          if (f == --l) return f;
          if (p(*l)) break;
     } // l points to a good element

     VALUE_TYPE(I) tmp;
     move(*f, tmp); // hole at f needs a good element

     while (true) {
          move(*l, *f);
          // fill the hole at f with good element at l
          // the hole is at l and needs a bad element
          do {
               if (++f == l) goto exit;
          } while (p (*f));
          // f points to a bad element
          move(*f, *l);
          // fill the hole at l with bad element at f
          // the hole is at f and needs a good element
          do {
               if (f == --l) goto exit;
          } while (!p (*l));
          // l points to a good element
     }
exit:
     // both f and l are equal and point to a hole
     move(tmp, *f);
     return f;
}
```

This piece of code is "optimal" in terms of many operations: it does the minimal number of comparisons, the (almost) minimal number of moves, the minimal number of iterator increments and iterator comparisons.

Problem: Find a case when the "optimal" algorithm would do one extra move [Joseph Tighe].

Problem: Find a way of avoiding an extra move [keep explicit track of the hole].

Problem: Use the same techniques to reduce the number of moves in `partition_forward`.

It should be noted, however, that in practice – or at least in practice as it is in 2005 – optimizing the number of moves does not significantly speed up the code for most types of elements. While we consider swap to be equivalent to three moves, for most modern computers it appears to be more accurate to consider swap to be equivalent to two loads followed by two stores, while move to be equivalent to one load and one store. If we switch to this system of accounting, we observe that `partition_bidirectional` does (almost) the same number of memory operations as `partition_bidirectional_minimal_moves.` It is a worthwhile thing to learn many of the optimization techniques, because of the twofold reason:

- optimization techniques are based on fundamental properties of algorithms that we study and allow us to understand the algorithms better;
- optimizations that are not applicable now in some domain will often become applicable again in a different domain.

Using sentinels

If we look at the code of the `partition_bidirectional_2` we observe that we do one iterator comparison for every predicate application – or almost one since the last iterator comparison during the running of the algorithm is not followed by a predicate application. If we know that our range contains both good and bad elements we can implement a function that will be trading an extra predicate call for a linear number of extra comparisons. If there is a bad element in the range we can always look for the first bad element from the left by writing:

```
while (p(*f)) ++f;
```

and be certain that after we stop all the elements in the range `[f0, f)` are going to satisfy the predicate and `f` will point to a bad element. We can now look for the good element from the right:

```
while (!p(*--l));
```

and be equally certain that we will stop at a good element. It is very easy to see that the only way they can cross is by one position only. That is, if they crossed then `f` is going to be the successor of `l`. (That, of course, presupposes that the predicate is truly functional and returns the same value when applied to the same element twice.)

That allows us to eliminate an iterator comparison from the inner loops:

```
template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I partition_bidirectional_unguarded(I f, I l, P p)
{
     assert(!all(f, l, p) && !none(f, l, p));
     while(true) {
          while (p(*f)) ++f;
          while (!p(*--l));
          if (++l == f) return f;
          swap(*f++, *--l);
     }
}
```

And that allows us to construct a new version of partition that first finds guards or sentinels on both sides and then calls the unguarded partition:

```
template <typename I,  // I models Bidirectional Iterator
          typename P>  // P models Unary Predicate
I partition_bidirectional_optimized(I f, I l, P p)
{
     f = find_if_not(f, l, p);
     l = find_backward_if(f, l, p);
     if (f == l) return f;
     swap(*f, *--l);
     return partition_bidirectional_unguarded(++f, l, p);
}
```

It is possible to eliminate extra iterator comparisons by also inlining finds and using the sentinel technique to trade a couple of applications of predicate for (potentially) linear number of iterator comparisons. It is, however, not an urgent optimization since if we assume that both good and bad elements are equally probable and that our input sequences are uniformly distributed then the number of extra iterator comparisons is going to be small.

Problem: What is the worst case number of the extra iterator comparisons in `partition_bidirectional_optimized`?

Problem: What is the average number of the extra iterator comparisons in `partition_bidirectional_optimized`?

Problem: Re-implement `partition_bidirectional_optimized` to minimize the number of iterator comparisons.

Problem: Combine the sentinel technique and the minimal moves optimization in a single algorithm.

Project: Measure the performance of all the partition algorithms that we have studied so far. Vary the element sizes from 32 bit integers and doubles all the way to structures with 64 byte size. Also use two different predicates: one which is inlined and very simple and the other one which is passed as a pointer to function. Come up with a recommendation on which of the algorithms are worth keeping in a library.

Project: Write a simple guide that will tell a user how to select a correct partition algorithm for the job.

Project: Write a library function that will correctly choose which of the partition algorithms to use depending on iterator requirements and, potentially, element size and properties of the predicate.

Partition copy

While it is often important to be able to partition a range in place, it is sometimes equally important to partition elements while copying them into a new place. It is, of course, often possible to accomplish it by first doing copy and then partition. There are two problems with this approach: the performance and the generality.

As far as the performance goes, we will need more than N moves. It would be terribly nice if we can accomplish our task with N moves only. The second problem is that in order to do copy first and partition afterwards we need to be able to traverse the resulting range again. And that means that we cannot use output iterators as a requirement for the destination. As a matter of fact the algorithm that is both minimal in terms of number of operations and absolutely minimal in terms of the requirements on the iterators for the result is so simple that it does not need any explanations. Go through the input range element by element sending good elements to one destination stream and the bad ones to a different one.

It is obvious how to start writing the algorithm:

```
if (p(*f))
     *r_g++ = *f++; // good result
```

```
else
     *r_b++ = *f++; // bad result
```

Footnote: This is, of course, “boy scout” code. Old C++ programmers will write without a blink something like:

```
(p(*f) ? *r_g++ : *r_b++) = *f++;
```

or , even more cryptic,

```
*(p(*f) ? r_g : r_b)++ = *f++;
```

The only remaining problem is to figure out what to return. And since the destinations of good and bad elements are different we have to return the final state of both:

```
template <typename I,  // I models Input Iterator
          typename O1, // O1 models Output Iterator
          typename O2, // O2 models Output Iterator
          typename P>  // P models Unary Predicate
pair<O1, O2> partition_copy(I f, I l, O1 r_g, O2 r_b, P p)
{
     while (f != l) {
          if (p(*f))
               *r_g++ = *f++;
          else
               *r_b++ = *f++;
     }
     return make_pair(r_g, r_b);
}
```

When we treat the stable partition algorithm we will rely on the fact that partition_copy is *stable*, that is, the relative order of among good elements is preserve and so is the relative order among the bad elements.

And, as we shall see later, it is often useful to have a move version of `partition_copy`:

```
template <typename I,  // I models Input Iterator
          typename O1, // O1 models Output Iterator
          typename O2, // O2 models Output Iterator
          typename P>  // P models Unary Predicate
pair<O1, O2> partition_move(I f, I l, O1 r_g, O2 r_b, P p)
{
     while (f != l) {
          if (p(*f))
               move(*f++,*r_g++);
          else
```

```
                move(*f++,*r_g++);
     }
     return make_pair(r_g, r_b);
}
```

Partitioning node-based structures

As with `reverse` and `rotate` it is sometimes desirable to have a different algorithm for node-based structures. If we transform the structure so that every node keeps its value, but all the nodes with correspondingly good or bad elements are linked together, then old node iterators will maintain their element, but they will be re-linked, correspondingly, into two different linked structures.

There is a standard technique for dealing with accumulating nodes: accumulating them in reverse order. Let us assume that `r_g` points to the all good nodes that we have already accumulated and `r_b` to all the bad ones. And we also know that f points to a node that we have not yet examined. Then we can see the inner part of our algorithm:

```
if(p(*f)) {
     set_successor(f, r_g);
     r_g = f;
else {
     set_successor(f, r_b);
     r_b = f;
}
```

Now, we added one more element to the appropriate structure. The problem is that we cannot get to the “old” successor of first. Well, that problem can be easily solved by saving it first. And that gives us the following implementation:

```
template <class I, // I models Forward Node Iterator
          class P> // P models Unary Predicate
pair<I, I> partition_node_reversed(I f, I l, P p)
{
     I r_g = l;
     I r_b = l;
     while (f != l) {
          I n = successor(f);
          if(p(*f)) {
               set_successor(f, r_g);
               r_g = f;
          else {
               set_successor(f, r_b);
               r_b = f;
          }
          f = n;
     }
     return make_pair(r_g, r_b);
}
```

It does N predicate applications, N `set_successor` operations and N `successor` operations: clearly minimal for predicate application and successor. And N `set_successor` operations is clearly only one greater than the worst case.

Problem: What is the worst case input for any algorithm for partitioning nodes structures if we count only `set_successor` operations?

Problem: What is the minimal expected number of `set_successor` operations that any algorithm for partitioning node structures will need assuming that good and bad elements are equally likely and distributed uniformly?

Now let us try to address the issue of minimizing the number of `set_successor` operations. (While solving this problem we will also solve the problem of making node partition stable, that is, assuring that good elements and bad elements are linked in the same order as they were in the original range.) It is pretty clear that we only need to change successor of a good element if the successor is bad and the other way around.

As a first step to construct the middle of such an algorithm, let us assume that somehow we obtained two iterators to the tail ends of good and bad elements called t_g and t_b. We can the proceed to construct both structures in the right order:

```
while (++f != l) {
     if (p(*f)) {
          set_successor(t_g, f);
          t_g = f;
     } else {
          set_successor(t_b, f);
          t_b = f;
     }
}
```

Now let us observe that we are doing too many `set_successor` operations. For all we know, `t_g` might already point to `f`; after all we came to `f` either from it or from `t_b`. and that we keep a flag `was_good` that indicates if the previous element we examined was good or bad:

```
while (++f != l) {
     if (p(*f)) {
          if (!was_good) {
               set_successor(t_g, f);
               was_good = true;
          }
          t_g = f;
     } else {
          if (was_good) {
               set_successor(t_b, f);
               was_good = false;
          }
          t_b = f;
     }
}
```

Now we are re-linking only the nodes that have successors of different "polarity"; if the successor of a good element is good the element keeps its successor; the same is true for bad elements. Note that we do not need to save the successor of `f`, since instead of `f` pointing to the appropriate substructure, the substructure gets to point to it.

There is an alternative to using a flag. We can duplicate the code for the loop one section for the case when the previous element was good and one for the case when the previous element was bad and then jump to the other section if the predicate value changes:

```
good:
     do { t_g = f;
          if (++f == l) goto exit;
     } while (p(*f));
     set_successor(t_b, f);
bad:
     do { t_b = f;
          if (++f == l) goto exit;
     } while (!p(*f));
     set_successor(t_g, f);
     goto good;
```

Now there are only two questions left: what to put after this code and what to put before. Let us start with the somewhat easier question of what to put after this code. Now we know that all the nodes are properly linked. We could also surmise that the tail end elements `t_g` and `t_b` correspond to some head elements `h_g` and `h_b`. So as a first approximation we can assume that our program ends with:

```
return make_pair(h_g, h_b);
```

But there is a little glitch with this ending: we just threw away the tail ends of both linked structures. And the client of our program may want to add more things to the tails. That, of course, is easily fixable, by replacing return statement with:

```
return make_pair(   make_pair(h_g, t_g),
                    make_pair(h_b, t_b));
```

It should be noted that if there are no good elements in the sequence the first pair will be `(l, l)`, if there are no bad elements the second pair will be `(l, l)`, and, finally, if the tail of either good or bad elements is not equal to `l`, the successor of the tail is not defined. We could have opted for always setting successor of such tails to last, but decided against it, since usually the head and tail nodes will have to be connected to a list header or spiced into a list. (We will fully understand it in the part of the notes dedicated to node-based containers.)

Now we know what should be the beginning of our algorithm. Before we get into the main loop, we should find h_g and h_b: the head nodes of the good list and the bad list. It is obvious that either one of them (or even both of them) might not exist. That raises a question what to return in such a case. The answer is self-evident: we can return a pair `make_pair(make_pair(l, l),make_pair(l, l))`. What we need to do is to find `h_g`, `h_b`, `t_g`, and `t_b`.

Now we can write the whole algorithm:

```
template <typename I, // I models Forward Node Iterator
          typename P> // P models Unary Predicate
pair<pair<I, I>, pair<I, I> > partition_node(I f, I l, P p)
{
     I h_g = l;
     I h_b = l;
     I t_g = l;
     I t_b = l;
     if (f == l) goto exit;
     if (!p(*f)) goto first_bad;
//   else        goto first_good;
first_good:
     h_g = f;
     do { t_g = f;
          if (++f == l) goto exit;
     } while (p(*f));
     h_b = f;
     goto current_bad;
first_bad:
     h_b = f;
     do { t_b = f;
          if (++f == l) goto exit;
     } while(!p(*f));
     h_g = f;
//   goto current_good;
current_good:
     do { t_g = f;
          if (++f == l) goto exit;
     } while (p(*f));
     set_successor(t_b, f);
//   goto current_bad;
current_bad:
     do { t_b = f;
```

```
          if (++f == l) goto exit;
     } while (!p(*f));
     set_successor(t_g, f);
     goto current_good;
exit:
     return make_pair(make_pair(h_g, t_g),
                      make_pair(h_b, t_b));
}
```

Remark: I am fully aware of Dijksra's strictures against using goto statement. For years I dutifully followed his dictum. Eventually, I discovered that on rare occasions I could write more elegant and efficient code if I used goto. In 1988 Dave Musser and I published a book containing a program with several goto statements. We received several angry letters explaining how ignorant we were. In spite of the criticism, I still maintain that goto is a very useful statement and should not be avoided if it helps to make the code cleaner. When I look at the previous piece of code, I find it beautiful. (Notice that I even added an unnecessary label `first_good` to make the code more symmetric, more understandable, and, yes, more beautiful. And I even added three unnecessary goto statements for the same reasons – but, not being sure that all the modern compilers eliminate goto from one address to the next, commented them out.)

It is easier to understand the algorithm if you view every label as a state and the goto-s as state transitions. In general, state machines are often easier to represent as labeled code sections with goto-s being the transitions.

Modern processors with their instruction level parallelism and predicated execution might not benefit at all from eliminating the `flag`. While we are certain of the pedagogical value of learning this transformation, it might not benefit the performance.

Problem: Implement partition_node using the flag and avoiding goto. Compare its performance with our version.

Problem: Implement a function `unique_node` that takes two iterators to a node structure and a binary predicate (defaulting to equality) and returning a structure with unique elements and a structure with "duplicates."

Finding the partition point

In the beginning of the chapter we discussed a problem of finding the partition point of an already partitioned range. There is an obvious solution:

```
find_if_not(f, l, p)
```

will definitely return the partition point of a partitioned range. The problem is that we will need to retest all good elements again.

It is easy to observe the following fundamental property of a partitioned range `[f, l)`: if an iterator `m` inside the range points at a good element then the partition point of `[f, l)` is located in the range `[successor(m), l)`; if m points at a bad element then the partition point is in the range `[f, m)`.

As far as an empty range goes, its beginning and its end both happen to be the partition points.

Let us assume for the moment that we are dealing with random access iterators and, therefore, can get to any element inside the range in constant time. If we have range represented as a pair of an iterator and an integer (the length of the range) and if we have a function `choose` that returns some non-negative integer less than the length of the range for any non-empty range, then for any such function `choose` there is a simple recursive algorithm for finding partition point:

```
template <typename I, // I models Random Access Iterator
          typename P> // P models Unary Predicate
I partition_point_recursive(I f, DIFFERENCE_TYPE(I) n, P p)
{
     if (n == 0) return f;
     N m = choose(n);
     if (p(*(f + m)))
          return partition_point_recursive(f + (m + 1),
                                             n – (m + 1));
     else
          return partition_point_recursive(f, m);
}
```

Since `0 ≤ m < n` we can be sure that both `n – m – 1` and `m` are less than `n` and not less than `0`; and, therefore, we can be sure that our program terminates. It is also obvious that a way of assuring that (no matter which path of the `if`-statement happens to be true) is by picking the choose function that for any positive `n` returns `n/2`.

Problem: Prove that picking `n/2` is indeed the best course.

Remark: If you are wondering if we are describing binary search, you are correct. But, as we shall see in the section on binary search (???), binary search is often defined incorrectly. It is essential to understand the interface and the implementation of the partition point finding algorithm to be able to handle binary search correctly. In particular, while it is self-evident what `partition_point` should return, it is far from self-evident what binary search should return. And even the great computer scientists often stumble defining (or even implementing!) it. This is why I believe that it is essential to deal thoroughly with predicate-based operations such as partition before attacking much more treacherous comparison-based operations.

Since our recursive calls are properly tail-recursive we can immediately obtain the following algorithm by resetting the variables in a loop instead of making a recursive call:

```
template <typename I, // I models Random Access Iterator
          typename P> // P models Unary Predicate
I partition_point_n_random_access
                         (I f, DIFFERENCE_TYPE(I) n, P p)
{
     while (n != 0) {
          if (p(*(f + n/2))) {
               f = (f + n/2) + 1;
               n = n - (n/2 + 1);
          } else {
          //   f = f;
               n = n/2;
          }
     }
     return f;
}
```

The algorithm does `ceiling(log(n))+ 1` predicate applications since we are reducing the length by dividing by 2 at every step.

What can we do if iterators which are given to us are less powerful than random access? While the efficiency of the algorithm will degrade dramatically, it is still quite useful in those cases when the predicate application is more expensive than the operation ++ on the iterators. If we use `find_if` to find partition point then on the expected cost of finding the partition point in a range of length `n` is

```
     c_linear = (n/2) * c_p + (n/2) * c_i.
```

where `c_p` is the cost of the predicate application and `c_i` is the cost of the iterator increment. (In other words, while doing linear search we expect to travel half of the way on the average.) If we use the partition_point_n algorithm the expected cost is going to be

```
     c_binary_best = (log(n) + 1) *c_p + n * c_i
```

since we are going to advance by `n/2`, `n/4`, `n/8`, etc. In those cases when the linked structure changes its size frequently we need to do another `n` increments and the cost becomes

```
     c_binary_worst = (log(n) + 1) *c_p + 2 * n * c_i
```

With large n we can safely ignore logarithmic terms and the binary algorithm wins against linear one when `c_p > c_i` if linked structure does not change its size and when `c_p > 3* c_i` if its size needs to be computed anew every time.

In practice, the cost of predicate application should be more than 4 times as expensive as iterator increment to really justify using binary search like algorithms on linked lists. Otherwise, it is usually better to use linear search. It usually means that if your predicate is a small inlined function object then using `find` is better; if it is a regular function call binary search like algorithms are better.

It is perfectly straightforward to modify our algorithm to work with forward iterators and we can do a few little optimizations as well:

```
template <typename I, // I models Forward Iterator
          typename P> // P models Unary Predicate
I partition_point_n(I f, DIFFERENCE_TYPE(I) n, P p)
{
     while (n != 0) {
          N h = n >> 1;
          I m = successor(f, h);
          if (p(*m)) {
               f = successor(m);
               n -= h + 1;
          } else {
               n = h;
          }
     }
     return f;
}
```

Problem: Implement a `partition_point` function that takes two iterators `[f, l)` as its arguments.

Operations on Function Objects

If we want to partition elements in a range of integers into those that are less than 5 and those that are not, we have several potential ways of doing it. We can implement a function that compares an integer with 5 and pass it to `partition`:

```
bool lt_5(int n) { return n < 5; }
int n[100];
// put some numbers into array
int* p = partition(n, n + 100, lt_5);
```

It will do the job, but it is going to be much slower than a hand-written code. Indeed we will have to have a function call (through a function pointer) where the hand-written code will have a single instruction comparing. Note, that declaring lt_5 to be an inline function will not change the situation since the code that compiler will generate for partition will be taking an arbitrary function pointer. The compiler generates separate instances of template functions only when the arguments types are different and the argument type of the pointer to our little function is the same as the type of any other pointer of the type `bool (*)(int)`. And that is the first reason for using function objects: different types for different function objects with same signatures allow the compiler to generate different instances of a template function and inline small function objects while doing it.

We can accomplish this by defining a single function object of an anonymous type:

```
struct
{
     bool operator()(int n) const {return n < 5;}
} less_than_5;
```

Now, it is easy to partition an array:

```
int n[100];
// put some numbers into array
int* p = partition(n, n + 100, less_than_5);
```

While it would be slightly nicer if we could just say something like:

```
partition(n, n + 100, function (int n) { return n < 5 })
```

and the language will create an anonymous function object and pass it to `partition`, we have to get used to the fact that no language – natural or artificial – can satisfy all of our requirements. (Think of all the controversies that were caused by the fact that English has no word to render Greek *anthropos,* Latin *homo,* or Russian *chelovek*.) We have to adapt to languages we have instead of always pining for the ideal language.

In any case, such a solution will not work if instead of 5 we need a predicate that is based on a value known only at run time. We will have to keep this value inside our function object. And, after all, that is one of the two main reasons for using function objects: the first is parameterizing generic algorithms by different pieces of code (that can often be inlined) and the second is providing a function (often inlined one) with its own persistent data.

We can solve the problem with the help of the following:

```
struct less_than_const_int
{
     int c;
     less_than_const_int(const int& n) : c(n) {}
     bool operator()(const int& x) const { return x < c; }
};
```

And it is an obvious candidate for templatization:

```
template <class T> // T models Strict Totally Ordered
struct less_than_const
{
     typedef T argument_type;
     typedef bool result_type;
     T c;
     less_than_const(const T& n) : c(n) {}
     bool operator()(const T& x) const { return x < c; }
};
```

Now, we can observe that even `operator<` can be abstracted away. And the same can be done for the last vestige of specificity: `bool`. As a matter of fact we can create a class that takes an arbitrary binary function object and binds its second argument to a value:

```
template <class F> // F models a Binary Function
struct binder2nd {
      typedef typename F::first_argument_type
                                          argument_type;
      typedef typename F::second_argument_type value_type;
      typedef typename F::result_type result_type;
      const F op;
      const value_type value;
      binder2nd(const F& x, const value_type& y)
            : op(x), value(y) {}
      result_type operator()(const argument_type& x)
      const
      {
            return op(x, value);
      }
};
```

Now we can accomplish our partition by calling:

```
int n[100];
// put some data into n
int a = 5;
partition(n, n + 100, binder2nd<less<int> >(
                         less<int>(), a));
```

It is possible to create a simple function that will allow us to save typing the name of the type of the binary function object twice:

```
template <class F, // F models binary function
          class T>
inline
binder2nd<F> bind2nd(const F& op, const T& x) {
      return binder2nd<F>(op,
                   typename F::second_argument_type(x));
}
```

And now we can call partition with:

```
partition(n, n + 100, bind2nd(less<int>(), a);
```

Problem: Implement `binder1st` and `bind1st`, which bind the first argument of a binary function object.

Binding is only one of several useful function object adaptors. Another useful function object operation is composition. It takes two function objects `f(x)` and `g(y)` and return a function object that does `f(g(y))`. It should be easy to see how to make such an adaptor:

```
template <class F, // F models unary function
          class G> // G models unary function
struct unary_compose {
      typedef typename G::argument_type argument_type;
      typedef typename F::result_type result_type;
      const F f;
      const G g;
      unary_compose(const F& x, const G& y) : f(x), g(y) {}
      result_type operator()(const argument_type& x) const
      {
            return f(g(x));
      }
};

template <class F, // F models unary function
          class G> // G models unary function
inline
unary_compose<F, G> compose1(const F& f, const G& g) {
    return unary_compose<F, G>(f, g);
}
```

Problem: Define a class `binary_compose` and a helper function `compose2` to be able to take a binary function object `f` and two unary function objects `g` and `h` and construct a binary function object that performs `f(g(x), h(y))`.

Footnote: Both compose1 and compose2 were included in HP STL. They were not, however, parts of the proposal and were not included in the C++ standard. I have no idea why they were not in the proposal. It is possible that somebody on the committee objected, or, it is possible that it was a result of my oversight.

Another adapter which we will eventually need is a `f_transpose` that takes a binary function object `f(x, y)` and returns a binary function object `f(y, x)`:

```
template <class F> // F models a Binary Function
struct transposer {
      typedef typename F::first_argument_type
```

```
                                               second_argument_type;
      typedef typename F::second_argument_type
                                               first_argument_type;
      typedef typename F::result_type result_type;
      const F f;
      transposer(const F& x) : f(x) {}
      result_type operator()(const first_argument_type& x,
                             const second_argument_type& y)
      const
      {
            return f(y, x);
      }
};

template <typename F> // F models Binary Function
inline
transposer<F> f_transpose(const F& f)
{
     return transposer<F>(f);
}
```

Problem: Implement classes `unary_negate` and `binary_negate` and the helper functions `not1` and `not2` that convert unary and binary predicates to their negations.

Remark: There are much more elaborate facilities in Boost for constructing function objects, but the their description does not belong to this book. As I stated before, I try to use as little of C++ as possible. While I needed to explain how to use and implement function objects, it is important to do it in a way that is totally transparent and does not use template metaprogramming techniques.

Stable partition

When people use both forward and bidirectional versions of partition algorithm they are sometimes surprised with the results. Let us consider a simple example of partitioning a sequence of integers:

```
0 1 2 3 4 5 6 7 8 9
```

with `is_even` as the predicate.

If we run `partition_forward` on this input we obtain:

0 2 4 6 8 5 3 7 1 9

While even numbers are in the same order as they were in the original sequence, the odd numbers are in total disarray.

In case of `partition_bidirectional` we see that both even and odd elements do not preserve their original order:

0 8 2 6 4 5 3 7 1 9

It is often important to preserve the original order of the good and bad elements. For example, imagine that a company is keeping a list of employees sorted by their last name. If they decide to partition them into two groups: US employees and non-US employees it would be important to keep both parts sorted; otherwise an expensive operation of sorting would be required.

Definition: The partitioning that preserves the relative ordering of both good and bad elements is called *stable* partitioning.

One of the important properties of stable partitioning is that it allows for *multipass* processing. Indeed if we need to partition a range `[f, l)` with a predicate `p1` and then partition the resulting sub-ranges with a predicate `p2` if we do it with a non-stable partition we need to write:

```
I m = partition(f, l, p1);
partition(f, m, p2);
partition(m, l, p2);
```

If, however, `stable_partition` is available the same goal can be accomplished with:

```
stable_partition(f, l, p2); // p2 before p1!
stable_partition(f, l, p1);
```

This property is very important when many passes are needed and the overhead of keeping track of small sub-ranges becomes difficult and expensive to manage. We shall see later how this property is used with remarkable effect in radix sorting. (pages ….)

Problem: Prove that stable partitioning of a given sequence with a given predicate is *unique*; that is, prove that there is only one permutation of a range that gives stable partitioning.

It is clear that we cannot implement `is_stably_partitioned` for an arbitrary type of elements the way we implemented `is_partitioned`. Indeed if somebody shows us a sequence:

0 4 2 1 3 5

we do not know if it is stable or not because we do not know what was the original order of the elements. It is, however, much easier to determine that one range is the stable partition of the second range than it is to determine if one range is a partition of the second range: uniqueness helps.

Indeed, in order for us to assure that an algorithm for partition works, we need to compare two sequences – the original one and the partitioned one. In order for the partition algorithm to be correct we need to assure two things: first, that the resulting sequence is partitioned and that is easy to test by applying `is_partitioned` function, and, second, that the resulting sequence is the permutation of the original one. And finding out if a sequence is a permutation of another sequence is difficult unless elements are totally ordered and we can reduce both sequences to a canonical form by sorting them. (We will have a further discussion of that when we deal with sorting.) If the only operation on the elements is equality, we do not know of an efficient way of determining if two sequences are permutations of each other.

Problem: Prove that determining if a sequence is a permutation of another requires O(n^2) operations if only equality of elements is available.

For small sequences we can determine if own is a permutation of the other with the help of a useful algorithm that goes through one range and attempts to find the equal element in the other. If elements are found they are moved up front. The algorithm returns if the first range is exhausted or when there is not equal element in the second:

```
template <typename I1,   // I1 models Input Iterator
          typename I2,   // I2 models Forward Iterator
          typename Eqv>  // Eqv models Binary Predicate
pair<I1, I2> mismatch_permuted(I1 f1, I1 l1,
                               I2 f2, I2 l2,
                    Eqv eqv = equal<VALUE_TYPE(I1)>())
{
     while (f1 != l1) {
          I2 n = find_if(f2, l2, bind1st(eqv, *f1));
          if (n == l2) break;
          swap(*f2++, *n);
          ++f1;
     }
     return make_pair(f1, f2);
}
```

(It should be noted that the second range is re-ordered to match the first. We should also remember not to use the algorithm for long ranges – it is quadratic.)

To determine if one range is a permutation of another we call `permuted_mismatch` and check if both ranges are exhausted:

```
template <typename I1, // I1 models Input Iterator
          typename I2> // I2 models Forward Iterator
inline
bool is_permutation(I1 f1, I1 l1, I2 f2, I2 l2)
{
     return mismatch_permuted (f1, l1, f2, l2) ==
               make_pair(l1, l2);
}
```

Problem: Assume that the elements in the range have a total ordering defined with `operator<` and implement a faster version of `is_permutation`.

Now we can produce a function that tests if the first range is the partitioning of the second:

```
template <typename I1, // I1 models Forward Iterator
          typename I2, // I2 models Forward Iterator
          typename P>  // P  models a Unary Predicate
bool is_partitioning(I1 f1, I1 l1, I2 f2, I2 l2, P p)
{
     return is_partitioned(f1, l1, p) &&
            is_permutation(f1, l1, f2, l2);
}
```

Now, in case of stable partition the testing is much easier to do. We need to go through the original range and check every element for equality with the corresponding element in the sub-range of the good elements if the original element is good and with the corresponding element from the sub-range of the bad elements otherwise. We can use a close analogue of the `mismatch` algorithm:

```
template <typename I1,   // I1 models Input Iterator
          typename I2,   // I2 models Input Iterator
          typename I3,   // I3 models Input Iterator
          typename P,    // P  models a Unary Predicate
          typename Eqv>  // Eqv models Binary Predicate
triple<I1, I2, I3> mismatch_partitioned(I1 f, I1 l,
                                        I2 f_g, I2 l_g,
                                        I3 f_b, I3 l_b,
                                        P p,
                         Eqv eqv = equal<VALUE_TYPE(I1)>())
{
     while (f != l) {
          if (p(*f)) {
               if (f_g == l_g || !eqv(*f, *f_g)) break;
               ++f_g;
          } else {
               if (f_b == l_b || !eqv(*f, *f_b)) break;
               ++f_b;
          }
          ++f;
     }

     return make_triple(f, f_g, f_b);
}
```

Now we can determine if a range is a stable partitioning of another range with the help of:

```
template <typename I1, // I1 models Forward Iterator
          typename I2, // I2 models Forward Iterator
          typename P>  // P  models a Unary Predicate
bool is_stable_partitioning(I1 f1, I1 l1,
                            I2 f2, I2 l2, P p)
{
     I1 m1 = find_if_not(f1, l1, p);
     return find_if(m1, l1, p) == l1 &&
               mismatch_partitioned(f2, l2, f1,
                         m1, m1, l1, p) ==
               make_triple(l2, m1, l1)
}
```

After we build all the machinery for testing stable partition, let us see what algorithms are available.

We observed before that the `partition_copy` algorithm is stable. That allows us to construct a simple stable partition algorithm that uses an additional buffer with n elements to partition a range:

```
template <typename I1, // I1 models Forward Iterator
          typename I2, // I2 models Forward Iterator
          typename P>  // P models Unary Predicate
I1 stable_partition_with_buffer_0(I1 f, I1 l, P p, I2 buf)
{
     pair<I1, I2> tmp = partition_copy(f, l, f, buf, p);
     copy(buf, tmp.second, tmp.first);
     return tmp.first;
}
```

This algorithm suffers from a potential inefficiency: it might need to copy large elements in the buffer and then to leave copies of the original data in the buffer. This is a clear case for using move semantics inside an algorithm:

```
template <typename I1, // I1 models Forward Iterator
          typename I2, // I2 models Forward Iterator
          typename P>  // P models Unary Predicate
I1 stable_partition_with_buffer(I1 f, I1 l, P p, I2 buf)
{
     pair<I1, I2> tmp = partition_move(f, l, f, buf, p);
     move(buf, tmp.second, tmp.first);
     return tmp.first;
}
```

As we shall see later, we also need to have a version that takes an iterator and a length as an argument:

```
template <typename I1, // I1 models Forward Iterator
          typename N,  // N models Integer
          typename I2, // I2 models Forward Iterator
          typename P>  // P models Unary Predicate
pair<I1, I1> stable_partition_n_with_buffer(
                              I1 f, N n, P p, I2 buf)
{
     triple<I1, I1, I2> tmp =
          partition_move_n(f, n, f, buf, p);
     move(buf, tmp.third, tmp.second);
     return make_pair(tmp.second, tmp.first);
}
```

Where `partition_move_n` is defined as:

```
template <typename I1, // I1 models Input Iterator
          typename N,  // N models Integer
          typename I2, // I2 models Output Iterator
          typename I3, // I3 models Output Iterator
          typename P>  // P models Unary Predicate
triple<I1, I2, I3> partition_move_n(
                         I1 f, N n, I2 r1, I3 r2, P p)
{
     while (n-- > 0) {
          if (p(*f))
               move(*f, *r1++);
          else
               move(*f, *r2++);
          ++f;
     }
     return make_triple(f, r1, r2);
}
```

While `stable_partition_with_buffer` is often sufficient in practice, in some cases there is not enough memory to accommodate the extra buffer of the same size as the range. To be able to handle cases like that we need to have an in-place algorithm that could partition the data while preserving stablity.

The easiest way for deriving such an algorithm for stable partition is to look again at the loop of the forward partition algorithm:

```
while (n != l) {
     if (p(*n)) {
          swap(*n, *m);
          ++m;
     }
     ++n;
}
```

The algorithm preserves the ordering of good elements. Every time we encounter a good element we put it right after the good elements encountered before. The algorithm could be called semi-stable. It is not so, however for bad elements. When we swap, the first bad element in the section of the bad elements encountered so far becomes the last bad element. Stability is lost. For example:

```
0 2 4 1 3 5 6
      ^     ^ ^
      f     n l
```

and we swap `1` and `6`. We need to preserve the run `1 3 5` in that order. Now, we spent quite some time studying a function that does just that. Instead of swapping we can rotate. `rotate(f, n, l)` will give us the desirable result:

```
0 2 4 6 1 3 5
```

That gives us a first draft of our stable partition:

```
template <typename I,  // I models Forward Iterator
          typename P>  // P models Unary Predicate
I stable_partition_slow(I f, I l, P p)
{
     I n = f;
     while (n != l) {
          if (p(*n)) {
               rotate(f, n, successor(n));
               ++f;
          }
          ++n;
     }
     return m;
}
```

While it works, it is quite slow. Since rotate is a linear time operation and it can be performed as many times as we encounter a good element, the complexity of the algorithm is quadratic. It is possible to modify the algorithm to find consecutive runs of good elements before doing rotate and reduce the number of operations by a constant, but it is not going to reduce complexity from quadratic to either linear or at least NlogN.

There is, however, a standard way to reduce the complexity by applying divide and conquer technique. If we split a range `[f, l)` into two equal (or almost equal) parts `[f, m)` and `[m, l)` and somehow manage to partition them in a stable manner:

```
G...GB...BG...GB...B
^    ^    ^    ^    ^
f    m1   m    m2   l
```

we can partition the whole range by rotating the range `[m1, m2)` formed by the partition points of the sub-ranges around the splitting point `m`.

And it is quite easy to stably partition an empty sequence or a sequence with one element.

That gives us a following algorithm:

```
template <typename I, // I models Forward Iterator
          typename N, // N models Integer
          typename P> // P models Unary Predicate
pair<I, I> stable_partition_inplace_n(I f, N n, P p)
{
      if (n == 0) return make_pair(f, f);
      if (n == 1) {
            I l = successor(f);
            if (p(*f)) l = f;
            return make_pair(f, l);
      }
      pair<I, I> i = stable_partition_inplace_n(
                           f, n/2, p);
      pair<I, I> j = stable_partition_inplace_n(
                           i.second, n - n/2, p);
      return make_pair(rotate(i.first, i.second, j.first),
                       j.second);
}
```

Footnote: As far as I know, this algorithm first appeared in the block algorithms section of USL C++ components Block Algorithms [Stepanov 1987].

Note how we use the divide and conquer not just to compute the partition point, but also to compute the mid-point – a potentially expensive operation for forward iterators. The first recursive call returns a partition point of a sub-problem and the beginning iterator of the second sub-problem. The second recursive call returns a partition point of a second sub-problem and the end of the range iterator for the problem itself.

And we can obtain a regular range interface by first computing the length of the range:

```
template <typename I, // I models Forward Iterator
          typename P> // P models Unary Predicate
inline
I stable_partition_inplace(I f, I l, P p)
{
      return stable_partition_inplace_n(f,
                                 distance(f, l),
                                 p).first;
}
```

It is clear that the algorithm has `ceiling(log(N))` levels and that only the bottom level does `N` predicate applications. Every other level does rotate `N/2` elements on the average, and, therefore, does somewhere between `N/2` and `3N/2` moves on the average depending on the iterator category. The total number of moves is going to be `Nlog(n)/2` for random access iterators and `3Nlog(n)/2` forward and bidirectional iterators.

Problem: How many moves will the algorithm perform in the worst case?

Now we have two versions of stable partitioning: one with buffer and one *in-place*. But in reality we need something in between: we need an algorithm that can use as much extra room as is available. The dichotomy between algorithms that use only polylogarithmic extra storage (in-place) and algorithms that can use as much as needed is useful to the inner world of the algorithmists, but it is of little practical utility. If we need to stably partition a million records it is more than likely that an extra buffer containing 10000 records is always available. Even a buffer containing 100000 records is usually not going to change the application performance. In other words, 1% is always available and 10% is frequently available even in the situations when memory is limited. It is, therefore, useful to introduce a different class of algorithms, *memory-adaptive* algorithms, that is, algorithms that improve their performance if more memory is available.

Our stable partition algorithm is an ideal candidate for making it into a memory-adaptive algorithm. If the data fits into a buffer, call `stable_partition_with_buffer`, otherwise use divide and conquer till it fits:

```
template <typename I, // I models Forward Iterator
          typename N, // N models Integer
          typename P, // P models Unary Predicate
          typename B> // B models Forward Iterator
pair<I, I> stable_partition_n_adaptive(I f, N n, P p,
                                       B b, N b_n)
{
      if (n == 0) return make_pair(f, f);
      if (n == 1) {
            I l = successor(f);
            if (p(*f)) f = l;
            return make_pair(f, l);
      }
      if (n <= b_n) return stable_partition_n_with_buffer(
                                                  f, n, p, b);
      pair<I, I> i = stable_partition_n_adaptive(
                           f, n/2, p, b, b_n);
      pair<I, I> j = stable_partition_n_adaptive(
                           i.second, n - n/2, p, b, b_n);
      return make_pair(rotate(i.first, i.second, j.first),
                         j.second);
}
```

We can obtain a regular range interface with:

```
template <typename I, // I models Forward Iterator
          typename N, // N models Integer
          typename P, // P models Unary Predicate
          typename B> // B models Forward Iterator
inline
```

```
I stable_partition_adaptive(I f, I l, P p, B b, N b_n)
{
      return stable_partition_n_adaptive(f,
                                         distance(f, l),
                                         p).first;
}
```

Problem: Measure the performance of `stable_partition_adaptive` when it is given a buffer of 1% , 10%, or 25% of the range size and compare it with performance of `stable_partition_inplace`.

In time critical applications it is important for the programmer to be able to do a careful allocation of memory resources and it is, therefore, important to provide an interface that allows for manual selection of the buffer. It is, however, often possible for a memory management system to figure out what is a proper buffer size for a given job. To enable programmers to obtain such temporary buffers STL defined a pair of template functions:

```
template <typename T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t);

template <typename T>
void return_temporary_buffer(T*);
```

The first function returns an optimal amount of memory now available which is not greater than the parameter to the function. The second function de-allocates the memory.

Footnote: It was my intention that system vendors will provide a carefully tuned function that will take into account the size of the physical memory, the memory available on the stack, etc, etc. I provided a temporary version that does call `malloc` with a given argument and if `malloc` returns 0, calls it with half the size, etc. I assumed that nobody will keep such a stupid code. It is interesting to note, that that is what the major vendors ship now (2005). In general, I have been trying to convince vendors and standard committees for quite some time now that it is essential to provide standard hooks to memory: cache structure, cache sizes, cache line sizes, physical memory size available to the process, stack size, size of available stack, etc, etc. So far I had no success. From all of that it follows that it was a mistake to include algorithms using temporary buffer into the standard. I should have insisted that the adaptive versions taking an explicit buffer were included. The present day wrappers such as we are going to see next are useless.

With a temporary buffer we can produce the following version of stable partitioning:

```
template <typename I, // I models Forward Iterator
          typename P> // P models Unary Predicate
I stable_partition(I f, I l, P p)
{
     ptrdiff_t n = distance(f, l);
     pair<VALUE_TYPE(I)*, ptrdiff_t> tmp =
          get_temporary_buffer<ptr_t, ptrdiff_t>(n);
     construct_any(tmp.first, tmp.first + tmp.second);
     I r = stable_ stable_partition_n_adaptive(
               f, n, p, tmp.first, tmp.second).first;
     destroy(tmp.first, tmp.first + tmp.second);
     return_temporary_buffer(tmp.first);
     return r;
}
```

Note that we need to initialize the buffer. Not that we need some particular values in it. We just need to have values into which we can move elements from the range. Any value of the element type will do. The annoying problem is that in many real cases we need to do nothing since any bit pattern will do just fine. This is why `C` and `C++` do not initialize arrays of built-in types: to avoid often unneeded operations. We attempt to accomplish this by introducing a function:

```
template <typename T>
void construct_any(T* f, T* l)
{
     T tmp;
     while (f != l) construct(&*f++, tmp);
}
```

where a single argument `construct` is a shorthand for a `C++` way of constructing a object in a previously allocated memory:

```
template <typename T>
inline
void construct(T* p, const T& x)
{
     new(p) T(x);
}
```

In order to avoid unnecessary initialization when we do not need it we define:

```
void construct_any(int* f, int* l)
{
}
```

for all the types such as `int` when we can safely accept any bit pattern.

Footnote: It is possible to extend a notion of `construct_any` to arbitrary types and by doing so regularize the special treatment of built-in types. It will require a rather simple addition to the language: the introducing a way of specifying a construct-any; calling it to initialize the arrays (and using it in the operator new); and, finally, to assure compatibility with existing code, defaulting construct_any to a default constructor. That would allow a user defined types to behave in the same way as built-in types and solve some existing performance issues which are caused by using user-defined types.

And we need to deal with destruction in a similar way:

```
template <typename I> // I models Forward Iterator
void destroy(I f, I l)
{
     while (f != l) destroy(&*f++);
}
```

And a single argument `destroy` is a convenient encapsulation of a peculiar `C++` way of explicitly calling destructors :

```
template <typename T>
inline
void destroy(T* p)
{
     p -> ~T();
}
```

Footnote: I made a mistake defining `construct` and `destroy` for the standard – they should have taken their arguments as references, not as pointers.

As with `construct_any`, if performance is needed, it is important to specialize destroy for those types for which destruction is not needed:

```
void destroy(int* f, int* l)
{
}
```

Footnote: I often hear "compiler will optimize it" statements. In 1995 I gave a talk at SGI – I was still working at HP Labs – and one of the members of the compiler organization assured me that their compiler will optimize away loops with unneeded destructions. Later that year when I joined SGI, I quickly discovered that the compiler did not optimize such loops. I spent 5 years as a member of SGI compiler team, and while some dramatic improvements were made in `C++` compilation during this time, but this particular problem was not fixed. And in 2005 it is still necessary to provide manual versions of functions like `construct_any` and `destroy` if one wants performance. Sadly enough, there are no serious efforts to develop a true high performance `C++` compilation system.

Reduction

When we implemented `stable_partition` we had to use the divide-and-conquer recursion. While it is often fine to use such recursion, we will now spend some time learning a general technique for eliminating it. While in practice it is needed only when the function call overhead caused by recursion starts effecting performance, the machinery for solving the problem is one of the most beautiful things in programming and needs to be learned irrespective of its utility.

One of the most important, most common loops in programming is a loop that adds a range of things together. The abstraction of such loop – it was introduced by Ken Iverson in 1961 – is called *reduction*. In general, reduction can be performed with any binary operation, but it is usually used with associative operations. Indeed, while

```
      ((...((a1 - a2) - a3)...) - an)
```

is a well defined expression, we seldom find a use for things like that. In any case, if an operation is not associative, we need to specify the order of evaluation. It is assumed that the default order of evaluation is the left-most reduction. (It is a natural assumption, since it allows us to reduce ranges with the weakest kind of iterators. Input iterators are sufficient.) It is an obvious loop to write. We set the result to the first element of the range and then accumulate elements into it:

```
     VALUE_TYPE(I) r = *f;
     while (++f != l)
          r = op(r, *f);
     return r;
```

The only problem is to figure out what to do for the empty range. One, and often useful solution, is to provide a version of reduction that assumes that the range is not empty:

```
template <typename I,   // I models Input Iterator
          typename Op>  // Op model Binary Operation
VALUE_TYPE(I) reduce_non_empty(I f, I l, Op op)
{
     assert (f != l);
     VALUE_TYPE(I) r = *f;
     while (++f != l)
          r = op(r, *f);
     return r;
}
```

But a general question remains. What is the appropriate value to return for an empty range? In case of an associative operation such as + it is commonly assumed that the right value to return is the identity element of the operation (`0` in case of +). Indeed, such a convention allows us to have the following nice property to hold. For any range `[f, l)`, for any iterator m inside the range and for any associative operation op on the elements of the range the following is true:

```
reduce(f, m, op) + reduce(m, l, op) == reduce(f, l, op)
```

In order for this to hold when m is equal to either f or l, we need reduce to return the identity element of the operation.

We can accomplish it quite easily with:

```
template <typename I,    // I models Input Iterator
          typename Op>   // Op model Binary Operation
inline
VALUE_TYPE(I) reduce(I f, I l, Op op,
               VALUE_TYPE(I) z = identity_element(op))
{
     if (f == l) return z;
     return reduce_non_empty(f, l, op);
}
```

Clients of the code need to provide either an explicit element to be returned for the empty range or the operation has to provide a way of obtaining its identity element. For some common cases we can provide standard solutions:

```
template <typename T>
inline
T identity_element(plus<T>) {
     return T(0);
}
```

A natural default for an additive identity element is a result of casting 0 into the element type. When the default does not work and it is easy to define a particular version of identity_element.

Problem: Define appropriate default identity_element for multiplies<T>.

Problem: Define appropriate default identity_element for:

```
struct min_int : binary_function<int, int, int>
{
     int operator()(int a, int b) { return min(a, b); }
};
```

If the reduction knows what the identity element is, it can do a standard optimization by skipping the identity elements in the range since combining the result with an identity element is not going to change it. This gives us a useful variation of reduce:

```
template <typename I,    // I models Input Iterator
          typename Op>   // Op model Binary Operation
```

```
VALUE_TYPE(I) reduce_nonzeros(I f, I l, Op op,
               VALUE_TYPE(I) z = identity_element(op))
{
      f = find_not(f, l, z);
      if (f == l) return z;
      VALUE_TYPE(I) r = *f;
      while (++f != l)
            if (*f != z)
                  r = op(r, *f);
      return r;
}
```

This version should be used when we want to avoid tests for identity elements inside the operation. In the cases when we have the code for the operation that handles only non-identity element cases, we do not then need to surround the code inside with two checks for identity (for the left and the right argument).

Now we can tackle the stable partition. First let us observe that if we have two sub-ranges `[f1, l1)` and `[f2, l2)` of a range `[f, l)` such that for some predicate `p`:

- `distance(f, l1) <= distance(f, f2)` – first sub-range is before the second
- `none(f1, l1, p) && none(f2, l2, p)` – both sub-ranges contain "bad" elements
- `all(l1, f2, p)` – and there are no "bad" elements in between them

then we can stably partition the combined range `[f1, l2)` by doing

```
rotate(f1, l1, f2)
```

and the result returned by the rotate is the partition point of the combined range. The following function object class performs the operation on such ranges:

```
template <typename I> // I models Forward Iterator
struct combine_ranges
     : binary_function<pair<I, I>, pair<I, I>, pair<I, I> >
{
     pair<I, I> operator()(const pair<I, I>& x,
                           const pair<I, I>& y) const
     {
          return make_pair(
                    rotate(x.first, x.second, y.first),
                    y.second);
     }
};
```

It is interesting to observe that we need to worry only about the sub-ranges containing "bad" elements. While we are combining the ranges of "bad" elements, the "good" elements bubble down to the front of the main range.

Problem: Prove that `combine_ranges` is associative.

We have an object to combine the ranges. It is a very simple to generate a sequence of trivial ranges containing "bad" elements. For every dereferenceable iterator in the main range we can produce a trivial sub-range with the help of the following:

```
template <typename I, // I models Forward Iterator
          typename P> // P models Unary Predicate
struct partition_trivial
     : unary_function<I, pair<I, I> >
{
     P p;
     partition_trivial(const P & x) : p(x) {}
     pair<I, I> operator()(I i) {
          I n = successor(i);
          return make_pair(p(*i) ? n : i, n);
     }
};
```

The only remaining problem is transforming a range of iterators to elements into a range of trivial ranges to be combined by `reduce` using `combine_ranges`. And that we can accomplish with the help of the following iterator-adaptor. It is constructed out of an incrementable object (an object with ++ defined on it) and a function object. When incremented, it increments the incrementable object. When dereferenced, it returns the result of an application of the function object to the incrementable object. It is a generally useful adapter:

```
template <typename I,      // I models Incrementable
          typename F = identity<I> >
                           // F models Unary Function
class value_iterator
{
public:
     typedef typename F::result_type value_type;
     typedef ptrdiff_t difference_type;
     typedef forward_iterator_tag iterator_category;
private:
     I i;
     F f;
public:
     value_iterator() {}
     value_iterator(const I& x, const F& y)
          : i(x), f(y) {}
     value_iterator& operator++() {
          ++i;
          return *this;
     }
     value_iterator operator++(int) {
          value_iterator tmp = *this;
          ++*this;
          return tmp;
     }
     value_type operator*() const {
          return f(i);
     }
     friend bool operator==(const self& a, const self& b) {
     //   assert(a.f == b.f);
     //        we comment the assert because unfortunately
     //        many function objects do not have == defined
          return a.i == b.i;
     }
     friend bool operator!=(const self& a, const self& b) {
          return a != b;
     }
};
```

Problem: Generalize `value_iterator` further by allowing a user to specify the meaning of ++ and providing a natural default for ++;

We can now obtain a slow version of stable partitioning by calling `reduce_nonzeros` with identity element equal the pair that is made of the last element of the range. (After all, the only place so far where it is going to be used is to be returned when the original range is empty. It is the right result in such a case. It is important to observe that for no dereferenceable iterator partition trivial will return such a range. Indeed, the empty ranges of "bad" elements returned by it are not identity elements!)

```
template <typename I, // I models Forward Iterator
          typename P> // P models Unary Predicate
I stable_partition_slow_iterative(I f, I l, P p)
{
     typedef partition_trivial<I, P> fun_t;
     typedef value_iterator<I, fun_t> val_iter;
     fun_t fun(p);
     pair<I, I> z(l, l);
     combine_ranges<I> op;
     val_iter f1(f, fun);
     val_iter l1(l, fun);
     return reduce_nonzeros(f1, l1, op, z).first;
}
```

Now, since we know that `combine_ranges` is associative, it is possible to replace left-most reduction with a balanced reduction that will apply the operation constructing a balanced tree.

That is, a tree that adds 4 elements like this:

```
  /\
 /\
/\
```

will be transformed into a tree that combines the same elements like that:

```
 /\
/\/\
```

The number of operations will remain the same, but the number of levels in the tree is going to be reduced. While reducing `n` elements with the left-most reduction requires `n-1` levels, doing it with the balanced reduction requires only `ceiling(log(n))` levels. And our `combine_ranges` belongs to a class of operation that work much better with the balanced reduction, namely, *linear-additive* operations. We will call an operation *linear-additive* if its cost is a linear function of the sizes of its arguments and the size of the result is the sum of the sizes of the arguments. It is easy to see that performing the left-most reduction with a linear-additive operation to a sequence of elements of the same size will require a O(n^2) cost while the balanced reduction will require O(log n). It is important to develop a generic version of the balanced reduction since we are going to encounter many algorithms where it can be useful.

Problem: Prove that `combine_ranges` is a linear-additive operation.

In order to implement the balanced reduction we need to observe that it needs to store up to log n intermediate results. The results can be stored in a simple counter where the k-th "bit" represents the sub-result of the balanced tree that resulted from reducing 2^k elements. The following procedure adds a new element to such a counter:

```
template <typename I,  // I models Forward Iterator
          typename Op> // Op models Binary Operation
VALUE_TYPE(I) add_to_counter(I f, I l, Op op,
                VALUE_TYPE(I) x,
                VALUE_TYPE(I) z = identity_element(op))
{
      if (x == z) return z;
      while (f != l) {
            if (*f != z) {
                  x = op(*f, x);
                  // op(*f, x) and NOT op(x, *f)
                  // because the partial result in *f
                  // is the result of adding elements
                  // earlier in the sequence
                  *f++ = z;
            } else {
                  *f = x;
                  return z;
            }
        }
        return x;
}
```

The procedure returns “zero” if the was a room in the counter to accommodate a new element or it returns an “overflow bit” if the last “bit” of the counter was combined into a new bit representing the reduction of 2^n elements where n is number of “bits” in the counter.

Now it is easy to produce an implementation of the balanced reduction. First we put all the elements from the input range into our counter. If the range size is a power of 2, we can obtain the result from the corresponding “bit” of the counter. If not, we need to reduce the counter. To minimize the amount of work we need to do a left-most reduction so that to combine “smaller” bits first, and we need to transpose the operation since when we combine two bits, the left one resulted from the elements that got into the counter after the elements which contributed to the right one and, therefore, their order needs to be exchanged:

```
template <typename I,  // I models Input Iterator
          typename Op> // Op models Binary Operation
void reduce_balanced(I f, I l, Op op,
                    VALUE_TYPE(I) z = identity_element(op))
{
     vector<VALUE_TYPE(I)> v;
     while (f != l) {
          VALUE_TYPE(I) tmp = add_to_counter(
                    v.begin(), v.end(), op, *f++, z);
          if (tmp != z) v.push_back(tmp);
     }
     return reduce_nonzeros(
                    v.begin(),v.end(), f_transpose(op), z);
}
```

Note that the `reduce_balanced` is not going to apply the operation to the identity element so that we do not need `reduce_non_zero_balanced`.

Footnote: It seems that many people independently discovered such iterative implementation of reduced_balanced. Knuth attributes it to McCarthy (Knuth …), but it seems to be just a pointer to the person who pointed it to him since no specific reference is given.

Problem: Why it is not important in this case that insertion into the back of a singly linked list takes linear time?

Finally we can now trivially obtain the balanced non-recursive implementation of `stable_partition_inplace` by replacing the call to the left-most reduction with the call to the balanced reduction:

```
template <typename I, // I models Forward Iterator
          typename P> // P models Unary Predicate
I stable_partition_inplace_iterative(I f, I l, P p)
{
     typedef partition_trivial<I, P> fun_t;
     typedef value_iterator<I, fun_t> val_iter;
     fun_t fun(p);
     pair<I, I> z(l, l);
     combine_ranges<I> op;
     val_iter f1(f, fun);
     val_iter l1(l, fun);
     return reduce_balanced(f1, l1, op, z).first;
}
```

Problem: Compare the performance of `stable_partition_inplace` with the performance of `stable_partition_inplace_iterative`. Explain the results.

In the `reduce_balanced` we are using vector as our counter. Both singly linked list and doubly linked list would have worked.

Problem: Implement versions of `reduce_balanced` using `list` and `slist`. Implement 3 different versions of `stable_partition_inplace_iterative` and compare their performances. A version of `slist` can be found at: http://stlab.corp.adobe.com/www/stldoc_Slist.html

Problem: Implement an iterative version of `stable_partition_adaptive` using `reduce_balanced`.

3-partition [Dutch National Flag – Dijkstra]

Some times the sequences with which we deal are divided into more than two kinds of elements. Before we address a problem of partitioning a range into an arbitrary number of buckets, let us spend some time on a very important case of partition, partition into three categories.

The algorithm for the three-way partitioning is commonly known a Dutch National Flag algorithm for the three colors: red, white and blue of the flag of the Kingdom of Netherlands. I do not know who introduced it first; I – as well as most other people – learned about it from an important book of Edsger Dijkstra *Discipline of Programming* (Dijkstra 1976). In it Dijkstra acknowledges his indebtedness for the problem to W. H. J. Feijen.

Remark. It is a very sad fact that the work of Edsger Dijkstra is becoming totally unknown to a modern programmer. While many of Dijkstra's opinions are extreme and one should take most of his pronouncements with a grain of salt, his work is central to programming as a scientific discipline and I would urge every young (or not so young) programmer to study his work. We should be grateful to the Computer Science Department of the University of Texas, Austin for creating the Internet archive of Dijkstra's works.

Instead of colors we are going to use integers; in particular, we assume that instead of a predicate returning a Boolean value – as in partition – we are given a *key function* that returns three values: `{0, 1, 2}` known as *keys*. Now we consider the range to be partitioned 3-ways if it contains no elements with keys `1` and `2` before elements with key `0`, and no elements with key `2` before elements with key `1`. It is very easy to implement a function to check if a range is partitioned:

```
template <typename I, // I models Forward Iterator
          typename F> // F models Unary Function
bool is_partitioned_3way(I f, I l, F key)
{
     equal_to<int> eq;
     f = find_if_not(f, l, compose1(bind2nd(eq, 0), key));
     f = find_if_not(f, l, compose1(bind2nd(eq, 1), key));
     f = find_if_not(f, l, compose1(bind2nd(eq, 2), key));
     return f == l;
}
```

Problem: Prove that `is_partitioned_3way` does what it claims to do.

It is important to observe that a different way of stating that a range is partitioned 3-way is by saying that the key function will return non-decreasing sequence of values or that if we assume that we have a function `is_sorted` (which we will indeed define in the Section …) we can check the range for being partitioned 3-way by the following simple function:

```
template <typename I, // I models Forward Iterator
          typename F> // F models Unary Function
bool is_partitioned_3way(I f, I l, F key)
{
     return is_sorted(
               f, l, compose2(less<int>(), key, key));
}
```

As a matter of fact, we can use the same code to verify n-way partitioning:

```
template <typename I, // I models Forward Iterator
```

```
          typename F> // F models Unary Function
bool is_partitioned_n_way(I f, I l, F key)
{
     return is_sorted(
               f, l, compose2(less<int>(), key, key));
}
```

(We will encounter an almost identical – but more general – function in the Section … as `is_sorted_by_key`.)

That shows us that there is a profound connection between sorting and partitioning. Indeed we can always implement an n-way partition (for any n, but 2 – see the remark at the end of the section) by implementing:

```
template <typename I, // I models Forward Iterator
          typename F> // F models Unary Function
pair<I , I> partition_n_way(I f, I l, F key) {
     sort(f, l, compose2(less<int>(), key, key);
}
```

(And we will also see a very similar function later under a different name.)

It is, of course, not a very interesting thing to do for small value of n since it is an NlogN algorithm for a linear time problem. It is much better, as we shall see when we study sorting, to implement sorting in terms of partitioning.

Now, let us get back to 3-way partition and Dijkstra's algorithm. Let us assume that somehow we managed to solve the problem up to some middle point `s`:

```
0000001111?????22222222
      ^   ^    ^       ^
      f   s    t       l  (first, second, third, last)
```

If `s` points to an element with key 1 we just advance `s`. If it is 0 we swap it with an element pointed at by `f` and advance both `f` and `s`. If it is 2 we decrement `t`; swap elements pointed by `t` and `s` and increment `s`. This algorithm works exactly like Lomuto's `partitition_forward` for 0 and 1, but sends 2 to the other end of the range.

The code looks like:

```
template <typename I, // I models Bidirectional Iterator
          typename F> // F models Unary Function
pair<I , I> partition_3way_bidirectional(I f, I l, F fun)
{
     I n = f;
     while (n != l) {
          int key = fun(*n);
          if (key == 0)
               swap(*f++, *n++);
          else if (key == 2)
               swap(*--l, *n);
          else
               ++n;
     }
     return make_pair(n, l);
}
```

It is clear that the algorithm does N predicate application and N swaps in the worst case and 2N/3 swaps on average.

Now, let us find an algorithm that allows us to do the 3way partition with forward iterators. Such an algorithm can be easily obtained using our standard inductive technique. Let us assume that somehow we managed to solve the problem up to some middle point:

```
000000111122222????????
      ^   ^    ^       ^
      f   s    t       l  (first, second, third, last)
```

Then we can partition it with:

```
template <typename I, // I models Forward Iterator
          typename F> // F models Unary Function
pair<I , I> partition_3way_forward(I f, I l, F fun)
{
     I t = f;
     I s = f;
     while (t != l) {
          int key = fun(*t);
          if (key == 0)
               cycle_left(*t, *s++, *f++);
          else if (key == 1)
               swap(*s++, *t);
          ++t;
     }
     return make_pair(f, s);
}
```

where `cycle_left` is defined as:

```
template <typename T>
inline
```

```
void cycle_left(T& a, T& b, T& c) // rotates to the left
{
     T tmp;
     move(a, tmp);
     move(b, a);
     move(c, b);
     move(tmp, c);
}
```

Remark. Edward Kandrot came up with an ingenious way of using a three argument `swap` to fix a major bug contained in the version of `partition_3way_forward` that I presented to my 2005 Adobe class and, at the same time to make the code more efficient. It is quite clear that it is useful to have versions of `cycle_left` up to some large (10?) number of arguments.

The algorithm does N predicate application and 2N swaps in the worst case and N swaps on average. In other words it does 30% more swaps on average than Dijkstra's `partition_3way_bidirectional`.

Project: Measure the performance of `partition_3way_forward` and `partition_3way_bidirectional` for different integral types (`char`, `short`, `int`, `long long`, etc) with a 3-way predicate that return a remainder of an integer divided by 3.

Problem: Implement `partition_4way_forward`.

Problem: Implement `partition_4way_bidirectional`.

Problem: What is the number of swaps that is performed by `partition_4way_forward` and `partition_4way_bidirectional` both in the worst case and on average?

Problem: Implement partiotion_copy_3way.

Problem: Implement stable_partiotion_3way.

`partition_3way` returns a pair of iterators which are two partition points. It is obvious that if a range is already partitioned we can find the partition points with the help of `partition_point_n`:

```
template <typename I, // I models Forward Iterator
          typename F> // F models Unary Function
pair<I , I> partition_point_3way_simple_minded
                               (I f, DIFFERENCE_TYPE(I) n, F fun)
```

```
{
     less<int> comp;
     return make_pair(partition_point_n(f, n,
               compose1(bind2nd(comp, 1), fun)),
                      partition_point_n(f, n,
               compose1(bind2nd(comp, 2), fun)));
}
```

The problem is that we are doing some extra work since both calls will repeat at least the first test of the middle element.

Problem: What is the largest number of duplicated tests?

We can easily fix that:

```
template <typename I, // I models Forward Iterator
          typename F> // F models Unary Function
pair<I , I> partition_point_3way
                          (I f, DIFFERENCE_TYPE(I) n, F fun)
{
     equal_to<int> eq;
     while (n > 0) {
          DIFFERENCE_TYPE(I) h = n>>1;
          I m = successor(f, h);
          switch (fun(*m++)) {
          case 0:
               f = m;
               n = n – h - 1;
               break;
          case 1:
               I i = partition_point_n(f, n – h - 1,
                         compose1(bind2nd(eq, 0), fun)),
               I j = partition_point_n(m, h,
                         compose1(bind2nd(eq, 1), fun));
               return make_pair(i, j);
          case 2:
               n = h;
          }
     }
     return make_pair(f, f);
}
```

Footnote: I have to admit an embarrassing thing: the interfaces for partition and partition_3way are not consistent. In 1986 I wrote my first library implementation of partition for a part of Ada Generic Library that could not be released since no Ada compiler would be able to compile deeply nested generics. (The fact that it could not be released is sad because in this part I used a concept of *coordinate* that later on turned into *iterator* in C++ STL. As far as I know, the code did not survive, except for, interestingly enough, code for the bidirectional partition which appeared in a paper in which Dave Musser and I introduced the term *generic programming* [Musser and Stepanov 1988].) In any case, I had to decide which way the partition is going to place the results: the elements satisfying the predicate before the elements not satisfying it, or the other way around. I decided that it is better to put "good" elements first since I could not see any particular reason for the opposite and both possible solutions seemed to be equivalent. When I was defining partition for STL in 1993, I did not question my prior reasoning and partition, again, moved good elements in front. It took another 10 years for me to see that I was wrong. When I started considering algorithms for 3-way, 4-way and n-way partitioning, I realized that it is really important that partition assures that the result is sorted according to partition key – the result of the key function. And all of the STL sorting algorithms assumed ascending order. Moreover, it would have allowed the following nice property to hold: partition_3way would have worked just like regular partition if given a two-valued key function returning {0, 1}. Moreover, sort with a comparison based on key-compare would have done partitioning – which is not true now for regular 2-way partition. (The problem will become even more visible in the next section on `partition_n_way`.) If you ever get a chance to design a new library (STL for C## ?) I encourage you to consider if you should fix the interface of partition by putting the bad elements first.

N-way partition

The idea of a 3-value predicate naturally generalizes to an idea of an *n-valued predicate*. A function is called an n-valued predicate if it returns an integral value in the range [0, n). In general, we can have n-valued predicates of any number of arguments. In this chapter, however, we restrict ourselves to unary predicates.

Most algorithms that deal with n-valued predicates need to know the value of n for a particular predicate. We call this value a *range size* of the predicate. That raises a design question of finding the range size for a given predicate. We could require that any such predicate `p` provides a member function `p.range_size()` or an ordinary function `range_size(p)`. Unfortunately, that would make it impossible to use our algorithms with pointers to functions. It will also require that we build special machinery for combining the code of the predicate with the integer that represents its range size. All the algorithms that deal with n-valued predicates will be passed the range size either explicitly or implicitly through other parameters.

We can easily generalize `partition_3way_forward` to deal with n-valued predicates. We need to replace 3 iterators that point to the end of a sub-range that contains all the elements with a corresponding value of the predicate with a range of n iterators. Following a well-established tradition we will call the sub-ranges *buckets*. (Knuth uses *piles*, but I find buckets to be a more generally accepted term.) So we are given a range – described as a pair of random access iterators `[f_b, l_b)` – to keep the iterators in the range that we are partitioning. (In this section `f_b` and `l_b` stand correspondingly for *first bucket* and *last bucket*. The last bucket, of course, points past the last "real" bucket.) For every integer in the range `[0, l_b - f_b)`, `*(f_b + i)`, is an iterator into the sequence being partitioned and the sub-range [`*(f_b + i), *(f_b + i + 1))` contains all of the elements with the predicate value `i` that have been discovered up till now. When we discover a new element with value `i`, all the buckets before the `i`-th bucket remain unchanged, the `i`-th bucket grows by one element and the buckets after the `i`-th bucket are shifted by one:

```
template <typename I, // I models Forward Iterator
          typename F, // F models N-Value Unary Predicate
          typename R> // R models Random Access Iterator
void partition_n_way(I f, I l, F fun, R f_b, R l_b)
// I is the value type of R
// the range of key is [0, l_b - f_b)
{
     fill(f_b, l_b, f);
     while (f != l) {
          VALUE_TYPE(I) tmp;
          move(*f++, tmp);
          R i_b = f_b + fun(tmp);
          R j_b = l_b;
          while (--j_b != i_b)
               move(**(j_b - 1), *(*j_b)++);
          move(tmp, *(*j_b)++);
     }
}
```

Problem: It is not strictly speaking necessary for the sequence of buckets in this algorithm to be randomly accessible. Change the code of `partition_n_way` so that the buckets are stored through weaker than random access iterators. (In most reasonable situation the buckets will be stored in an array or a vector.)

Problem: Notice that `partition_n_way` does some unnecessary moves. It saves and restores the tested element even when it belongs to the last bucket and the only thing needed is to increment the corresponding bucket pointer. It also does unnecessary moves when it goes through empty buckets. Implement a version of it with fewer moves.

If N is the length of the range being partitioned and k is the range size of the predicate then the algorithm always does N predicate application and N(k+1) moves in the worst case and N(k+1)/2 moves on average. The next problem will show a way of reducing the number of moves by a factor of 2 for a case of bidirectional iterators. But even such a reduction will not make this algorithm reasonable when N gets large. In practice, it is almost never worthwhile to use it: as we shall see there are much better alternatives.

Problem: Implement a version of `partition_n_way` for bidirectional iterators that – similar to `paritition_3way_bidirectional` – will construct half the buckets on the left and half the buckets on the right of the ever decreasing range `[f, l)`. (Mark Ruzon.)

The next set of algorithms is built on the notion of buckets which we just encountered in `partition_n_way`. The fundamental ideas go back to the MIT master thesis of Harold H. Seward [Seward, 1954] which is definitely one of the most important early computer science papers. Seward introduced the partitioning algorithms in the context of a new sorting algorithm that he introduced: radix sort, which we will describe later in the section.

The fundamental idea behind "bucket" algorithms is that we can use a range of buckets containing positions where we put the next element with a given value of the key function:

```
template <typename I, // I models Input Iterator
          typename F, // F models N-Value Predicate
          typename R> // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
// VALUE_TYPE(I) == VALUE_TYPE(VALUE_TYPE(R))
void bucket_copy(I f, I l, F fun, R f_b)
{
     while (f != l) {
          *(f_b[fun(*f)]++) = *f;
          ++f;
     }
}
```

We, of course, rely on the assumption that the range size of `fun` is the same as the size of the range of buckets. The client of the code should know both and provide us with the right initial values in the buckets. We do not need to return anything since the buckets are updated and will contain updated iterators pointing to the end of corresponding ranges.

```
template <typename I, // I models Input Iterator
          typename F, // F models N-Value Predicate
          typename R> // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
void bucket_move(I f, I l, F fun, R f_b)
{
     while (f != l) {
          move(*f, *(f_b[fun(*f)]++));
          ++f;
     }
}
```

```
template <typename I, // I models Forward Iterator
          typename F, // F models N-Value Predicate
          typename R> // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
void bucket_swap(I f, I l, F fun, R f_b)
{
     while (f != l) {
          swap(*f, *(f_b[fun(*f)]++));
          ++f;
     }
}

template <typename I, // I models Forward Node Iterator
          typename F, // F models N-Value Predicate
          typename R> // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
// VALUE_TYPE(R) == pair<I, I>
void partition_bucket_node(I f, I l, F fun, R f_b, R l_b)
{
     fill(f_b, l_b, make_pair(l, l));
     while (f != l) {
          R c_b = f_b + fun(*f);
          if ((*c_b).first == l)
               (*c_b).first = f;
          else
               set_successor((*c_b).second, f);
          (*c_b).second = f++;
     }
}

template <typename I, // I models Forward Node Iterator
          typename R> // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
// VALUE_TYPE(R) == pair<I, I>
pair<I, I> connect_bucket_nodes(I l, R f_b, R l_b)
{
     while (f_b != l_b && (*f_b).first == l) ++f_b;

     if (f_b == l_b) return make_pair(l, l);

     I f = (*f_b).first;
     I c = (*f_b).second;

     while (++f_b != l_b) {
          if ((*f_b).first != l) {
               set_successor(c, (*f_b).first);
               c = (*f_b).second;
          }
     }

     return make_pair(f, c);
}

template <typename I, // I models Forward Node Iterator
          typename F, // F models N-Value Predicate
          typename R> // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
// VALUE_TYPE(R) == pair<I, I>
pair<I, I> partition_bucket_node_connected
                    (I f, I l, F fun, R f_b, R l_b)
{
     partition_bucket_node(f, l, fun, f_b, l_b);
     return connect_bucket_nodes(l, f_b, l_b);
}


template <typename I, // I models Input Iterator
          typename F, // F models N-Value Predicate
          typename R> // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
// convertible_to(RESULT_TYPE(F), DIFFERENCE_TYPE(R))
void accumulate_histogram(I f, I l, F fun, R f_b)
{
     while (f != l) {
          ++f_b[fun(*f)];
          ++f;
     }
}

template <typename I1, // I1 models Forward Iterator
          typename I2, // I2 models Forward Iterator
          typename F,  // F models N-Value Predicate
          typename R>  // R models Random Access Iterator
// VALUE_TYPE(I1) == ARGUMENT_TYPE(F)
// convertible_to(VALUE_TYPE(I1), VALUE_TYPE(I2))
// VALUE_TYPE(R) == I2
void compute_buckets(I1 f, I1 l, I2 r, F fun,
                         R f_b, R l_b)
{
     vector<DIFFERENCE_TYPE(I2)> v(l_b - f_b);
     fill(v.begin(), v.end(), 0);
     histogram(f, l, fun, v.begin());
     vector<DIFFERENCE_TYPE(I2)>::iterator i = v.begin();
     while (f_b != l_b) {
          *f_b = r;
          advance(r, *i);
          ++f_b;
          ++i;
     }
}

template <typename I1, // I1 models Forward Iterator
          typename I2, // I2 models Forward Iterator
          typename F,  // F models N-Value Predicate
          typename R>  // R models Random Access Iterator
// VALUE_TYPE(I1) == ARGUMENT_TYPE(F)
// convertible_to(VALUE_TYPE(I1), VALUE_TYPE(I2))
// VALUE_TYPE(R) == I2
void bucket_partition_copy(I1 f, I1 l, I2 r, F fun,
                              R f_b, R l_b)
{
     compute_buckets(f, l, r, fun, f_b, l_b);
     bucket_copy(f, l, fun, f_b);
}

template <typename I,  // I models Forward Iterator
          typename F,  // F models N-Value Predicate
          typename R>  // R models Random Access Iterator
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
// VALUE_TYPE(R) == I
void bucket_partition(I f, I l, F fun, R f_b, R l_b)
{
     compute_buckets(f, l, f, fun, f_b, l_b);
     vector<I> b_e(l_b - f_b, l);
     copy(f_b + 1, l_b, b_e.begin());
     pair<R, vector<I>::iterator> p(f_b, b_e.begin());
     while (true) {
          p = mismatch(p.first, l_b, p.second);
          if (p.first == l_b) return;
          bucket_swap(*p.first, *p.second, fun, f_b);
     }
}
```

Remark: The idea for this algorithm was suggested to me by Lubomir Bourdev during 2005 class at Adobe after I rashly stated that such an algorithm cannot exist. The long years of knowing the fact that the radix sort cannot be done in-place prevented me from seeing that if stability is not required what is not possible becomes very possible. Knuth describes a similar algorithm in the problem 5.2-13 [Knuth 1972]. (The algorithm was later published by Burnetas, Solow and Agarwal in 1997 [Burnetas, 1997].) I am positive that I read the solution to Knuth 5.2-13 sometime in the ‘80s but completely forgot it.

```
template <typename I, // I models Forward Iterator
          typename F> // F models N-Value Predicate
void partition_n_way_recursive(I f, I l, F fun,
                               int i, int j)
{
     if (j - i <= 1) return;
     int h = i + (j - i)/2;
     I m = partition(f, l,
          compose1(bind2nd(less<int>(), h), fun));
     partition_n_way_recursive(f, m, i, h);
     partition_n_way_recursive(m, l, h, j);
}

template <typename I, // I models Forward Iterator
          typename F> // F models N-Value Predicate
void partition_n_way_recursive1(I f, I l, F key,
                                int i, int j)
{
     while (j - i > 1) {
          int h = i + (j - i)/2;
          I m = partition(f, l,
               compose1(bind2nd(less<int>(), n), key));
          partition_n_way_recursive1(f, m, i, h);
          f = m;
          i = h;
     }
}

template <typename T1, typename T2,
          typename T3, typename T4>
struct quadruple
{
     T1 first;
     T2 second;
     T3 third;
     T4 forth;
     quadruple(const T1& x,
               const T2& y,
               const T3& z,
               const T4& w)
          : first(x), second(y), third(z), forth(w) {}
};

template <typename T1, typename T2,
          typename T3, typename T4>
inline
```

```
void scatter(const quadruple<T1, T2, T3, T4>& x,
             T1& a, T2& b, T3& c, T4& d)
{
     a = x.first;
     b = x.second;
     c = x.third;
     d = x.forth;
}

template <typename T1, typename T2,
          typename T3, typename T4>
inline
void gather(quadruple<T1, T2, T3, T4>& x,
          const T1& a, const T2& b,
          const T3& c, const T4& d)
{
     x.first  = a;
     x.second = b;
     x.third  = c;
     x.forth  = d;
}
// need to define ==, != and the relational operators
// for quadruple

template <typename I, // I models Forward Iterator
          typename F> // F models N-Value Predicate
void partition_n_way(I f, I l, F fun, int n)
{
     typedef quadruple<I, I, int, int> quad;
     vector<quad> stack;
     int i = 0;
     int j = n;
     while (j - i > 1 || !stack.empty()) {
          if (j - i <= 1) {
               scatter(stack.back(), f, l, i, j);
               stack.pop_back();
          }
          int h = i + (j - i)/2;
          I m = partition(f, l,
               compose1(bind2nd(less<int>(), h), fun));
          if (h - i > 1)
               stack.push_back(quad(f, m, i, h));
          f = m;
          i = h;
     }
}
```

Radix sorting

```
template <typename I, // I models Forward Node Iterator
          typename F> // F models 256-way Predicate
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
pair<I, I> partition_node_byte(I f, I l, F fun,
                                 bool is_unsigned)
{
     pair<I, I> b[256];
     partition_bucket_node(f, l, fun, b, b + 256);
     pair<I, I> x = connect_bucket_nodes(l, b, b+128);
     pair<I, I> y = connect_bucket_nodes(l, b+128, b+256);
     if (x.first == l) return y;
     if (y.first == l) return x;
     if (!is_unsigned) swap(x, y);
     set_successor(x.second, y.first);
     return make_pair(x.first, y.second);
}

template <typename N> // N models Integer
class nth_byte
     : public unary_function<N, int>
{
private:
     int n;
public:
     nth_byte(int i) : n(i << 3) {}
     int operator()(const N& k) const {
          return int(k >> n) & 255;
     }
};

template <typename I, // I models Forward Node Iterator
          typename F> // F models Unary Function
                      // F::result_type models Integer
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
pair<I, I> radix_sort_node(I f, I l, F fun,
                         int f_lsb, int l_lsb, bool is_nn)
// f_lsb is the number of the first least significant byte
// to be considered
// l_lsb is the number of the last least significant byte
// to be considered
// is_nn is true if fun returns values >= 0
{
     typedef typename F::result_type N;
     pair<I, I> p(f, l);
     if (f == l) return p;
     for (int i = f_lsb; i <= l_lsb; ++i) {
          p = partition_node_byte(
                    p.first,
                    l,
                    compose1(nth_byte<N>(i), fun),
                    i < l_lsb || is_nn);
          set_successor(p.second, l);
     }
     return p;
}

template <typename N> // N models Integer
inline
bool is_unsigned()
{
     return N(0) - N(1) >= N(0);
}

template <typename I, // I models Forward Node Iterator
          typename F> // F models Unary Function
                      // F::result_type models Integer
// VALUE_TYPE(I) == ARGUMENT_TYPE(F)
inline
pair<I, I> radix_sort_node(I f, I l, F fun)
{
     typedef typename F::result_type Integer;
     return radix_sort_node(f, l, fun,
                            0, sizeof(Integer) - 1,
                            is_unsigned<Integer>());
}

template <typename T>
struct identity_function : unary_function<T, T>
{
     T operator()(const T& x) const {return x;}
};

template <typename I> // I models Forward Node Iterator
// VALUE_TYPE(I) models Integer
inline
pair<I, I> radix_sort_node(I f, I l)
{
     typedef typename VALUE_TYPE(I) Integer;
     identity_function<Integer> id;
     return radix_sort_node(f, l, id,
                                    0, sizeof(Integer),
                                    is_unsigned<Integer>());
}

template <typename N> // N models Integer
int actual_sizeof(N n)
{
     N mask = ~N(255);
     if (n < 0) n = -n;

     int k = 1;
     while (n & mask) {
          ++k;
          n >>= 8;
     }
     return k;
}
```

Historical note

The forward partition algorithm is due to Nico Lomuto who was looking for a simpler way to implement the quicksort inner loop [Bentley 1984]. Its main advantages over Hoare's algorithm are, firstly, that it works for forward iterators and, secondly, that it preserves the relative ordering of the elements that satisfy the predicate (see the discussion of stable partition). Its disadvantages are that it does more swaps on the average than the next algorithm and that it cannot be modified to split the range containing equal elements into two equal parts, which, as we shall see in the chapter on sorting [page ???], makes it really unsuitable for being used in quicksort – for which purpose it is, nevertheless, frequently recommended.

The bidirectional partition was introduced by C. A. R. Hoare as a part of his quicksort algorithm [Hoare 1961]. The algorithm for partition with the minimal number of moves was described by Hoare in his remarkable and unjustly forgotten paper [Hoare 1962] which also introduced the idea of using sentinels to minimize the number of operation in the inner loop. This paper, in my opinion, is a serious contender for the title of the best ever paper in Computer Science.

Using partition for implementing random shuffle for forward iterators was invented by Raymond Lo and Wilson Ho in 1997 during the SGI course on Generic programming.

Solutions to selected problems:

```
template <typename I,  // I models Forward Iterator
          typename P>  // P models Unary Predicate
I partition_forward_minimal_moves(I f, I l, P p)
{
     while (true) {
          if (f == l) return f;
          if (!p(*f)) break;
          ++f;
     }
     I n = f;
     while (true) {
          if (++n == l) return f;
          if (p(*n)) break;
     }
     VALUE_TYPE(I) tmp;
     move(*f, tmp);
     move(*n, *f);
     ++f;
     I h = n;
     while (++n != l)
          if (p(*n)) {
               move(*f, *h);
               move(*n, *f);
               h = n;
               ++f;
          }
     move(tmp, h);
     return f;
}


template <typename T,
          typename R> // R models Strict Weak Ordering on T
inline
bool compare_3way(const T& x, const T& y, R r = less<T>())
{
     if (r(x, y)) return -1;
     if (r(y, x)) return 1;
     return 0;
}

template <typename I, // I models Forward Iterator
          typename F, // F models N-Value Unary Predicate
          typename R> // R models Random Access Iterator
void partition_n_way_fewer_moves(I f, I l, F fun,
                                  R f_b, R l_b)
// VALUE_TYPE(R) == I
// the range of key is [0, l_b - f_b)
{
     fill(f_b, l_b, f);
     while (f != l) {
          R i_b = f_b + fun(*f);
          if (i_b == l_b - 1) {
               ++*i_b;
               ++f;
          } else {
               VALUE_TYPE(I) tmp;
               move(*f++, tmp);
               R j_b = l_b;
               while (--j_b != i_b) {
                    if (*predecessor(j_b)) != *j_b)
                         move(**predecessor(j_b), **j_b);
                    ++*j_b;
               }
               move(tmp, *(*j_b)++);
          }
     }
}
```

Proposition: Given two sequences of length *n* with only equality comparison, *O(n^2)* operations are required to determine if one sequence is a permutation of the other.

Proof:

Form a bipartite graph where the nodes are the elements of the sequences and each pair of elements from the two sequences is connected by an edge. Each edge in the graph represents an as yet un-compared pair of elements. Initially, the number of edges is *n^2*.

The algorithm to determine whether one sequence is a permutation of the other is modeled as sequence of edge tests in the graph. That is, it sequentially selects an edge in the graph and tests for equality of the two nodes. If the nodes are equal, they can be removed from the graph, as well as any edges incident on them. If the nodes are not equal, then only the tested edge is removed. The algorithm stops, either if all nodes are removed (indicating that one sequence is a permutation of the other), or if one of the nodes becomes disconnected (indicating that the one sequence is not a permutation of the other).

The proof is to show that any such algorithm must always do *O(n^2)* comparisons in the worst case.

Suppose that for every test where the degree of each of the two nodes is greater than one, the result of the comparison is that they are not equal. If the degree of either node is one, then the result of the comparison is that they are equal.

Suppose the algorithm proceeds from its initial state doing a sequence of comparisons until it reaches the first pair that is equal. Since the initial degree of each node in the graph is *n*, this will require at least *n + k* tests, where *k* is the number of comparisons that did not involve either of the two equal nodes. The two nodes are removed, as well as any edges incident on them. What remains is a bipartite graph of *2(n – 1)* nodes and *(n – 1)^2 – k* edges.

The tests to remove the extra *k* edges can equivalently be done immediately following the removal of the two equal nodes. That is, immediately following the removal of the two nodes we have a complete bipartite graph of *(n – 1)^2* edges, and then the *k* edges are removed. In this case, at least *n* tests are required to obtain the first equal pair and reduce the graph.

The reduced graph is a complete instance of the sub-problem for sequences of length *n-1*. Therefore we have the recurrence:

c(n) >= n + c(n-1)

where *c(n)* is the worst case number of comparisons for sequences of length *n*. Consequently, *c(n) = O(n^2).*

(The proof is contributed by Jon Brandt.)